NG RISC-V ASIC Design

SCALER: Side-Channel Leakage Reduction in RISC-V Core,
AKA
NG RAD: Next-Generation RISC-V ASIC Design

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
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

1 Executive Summary **1**

2 Project Description **2**
2.1 Background & Motivation 2
2.1.1 ASIC Design Process with EDA Tools and Cadence 2
2.1.2 Importance of Hardware Security 3
2.1.3 Types of Hardware Attacks 4
2.1.4 Spectre and Meltdown 7
2.1.5 RowHammer 8
2.1.6 RAMBleed 8
2.1.7 RowPress 9
2.1.8 Trojans 10
2.2 Existing Products and Work 10
2.2.1 INVITED: Protecting RISC-V against Side-Channel Attacks 10
2.2.2 PARAM: Power Attack Resistant Microprocessor 11
2.2.3 Secure RISC-V Microprocessor Based on SERV Architecture 12
2.3 Goals and Objectives 13
2.3.1 Basic Goals 13
2.3.2 Advanced Goals 14
2.3.3 Stretch Goals 14
2.3.4 Basic Objectives 14
2.3.5 Advanced Objectives 14
2.3.6 Stretch Objectives 15
2.4 Engineering Specifications Table 15
2.5 House of Quality 16
2.6 Hardware & Software Block Diagrams 17
2.7 Prototype Illustration 18

3 Research and Investigation **19**
3.1 ASIC Design Flow 19
3.1.1 Specifications 19
3.1.2 Microarchitecture 20
3.1.3 RTL Design 20
3.1.4 Verification 20
3.1.5 Implementation 21

3.1.6 Logic Synthesis . . . 21
3.1.7 Floorplanning . . . 22
3.1.8 Place and Route . . . 22
3.1.9 Static Timing Analysis . . . 22
3.1.10 Tapeout . . . 22
3.2 Verilog Hardware Description Language for ASIC Design . . . 23
3.2.1 Purpose of a Hardware Description Language . . . 23
3.2.2 Key features of Verilog . . . 23
3.2.3 Verilog in the Design Workflow . . . 24
3.2.4 Applications of Verilog . . . 24
3.2.5 Advantages and Limitations . . . 25
3.2.6 Importance of Verilog in the Industry . . . 25
3.2.7 RISC-V Core Selection . . . 26
3.3 EDA Tool and Technology Comparison . . . 30
3.3.1 Cadence Design Suite . . . 31
3.3.2 Synopsys Tools . . . 31
3.3.3 Mentor / Siemens EDA . . . 31
3.3.4 Open-Source Tools . . . 31
3.4 Ibex Verification Specifics - OpenTitan . . . 32
3.5 Verification Methodology Selection . . . 33
3.6 Randomness vs. Pseudo-randomness . . . 34
3.6.1 Linear Feedback Shift Registers (LFSRs) . . . 36
3.6.2 Ring-Oscillators . . . 37
3.6.3 Gaussian Number Generators . . . 38
3.6.4 Statistical Testing Standards for PRNGs . . . 39
3.7 Side-Channel Mitigation Techniques . . . 39
3.7.1 Masking Order . . . 40
3.7.2 Hiding Techniques . . . 45
3.7.3 Process Node . . . 49
3.7.4 Operating Voltage Range . . . 53
3.7.5 IR Drop . . . 54
3.7.6 Timing Failure . . . 55
3.7.7 Operating Frequency . . . 56
3.7.8 Instructions Per Cycle . . . 56
3.7.9 Operating Temperature Range . . . 58
3.7.10 Cache Size . . . 59
3.8 FuseSoC . . . 60
3.8.1 Introduction and Motivation . . . 60
3.8.2 Core Concepts and Motivation . . . 61
3.8.3 Workflow and Usage . . . 62
3.9 LiteX . . . 62
3.10 Bender (ETH Zürich) . . . 63
3.11 Chipyard . . . 64

4 Standards and Design Constraints . . . 67

4.1 Industry Standards . . . 67
4.1.1 Cryptographic Standards . . . 67
4.2 Design Standards . . . 70
4.2.1 SystemVerilog and RTL Coding Standards . . . 70
4.2.2 Low Power and Low Switching Activity Standards . . . 71
4.2.3 ASIC Synchronous Design Standards . . . 71
4.3 RISC-V Architecture . . . 72
4.3.1 Basic Architecture . . . 73
4.3.2 ISA Extensions . . . 74
4.3.3 Memory . . . 77
4.3.4 Pipelining . . . 78
4.3.5 Privileges . . . 79
4.4 Verification and Validation Standards . . . 79
4.4.1 Functional and Code Coverage . . . 79
4.5 Universal Verification Methodology (UVM) . . . 80
4.5.1 UVM Wrap-up . . . 81
4.6 OpenTitan . . . 82
4.6.1 Comportable IP (CIP) Architecture . . . 82
4.6.2 CIP Library Classes . . . 82
4.6.3 Security Verification in CIP Library . . . 83
4.7 Engineering Standards Selection . . . 83
4.7.1 Instruction Set . . . 83
4.7.2 Operating Temperature Range . . . 83
4.7.3 Operating Voltage Range . . . 83
4.7.4 Process Node . . . 84
4.7.5 IR Drop . . . 84
4.7.6 Average Frequency . . . 84
4.7.7 Masking Order . . . 84
4.7.8 Chip Density . . . 85
4.7.9 Cache Size . . . 85
4.7.10 Timing Failure Count . . . 85
4.8 Practical and Security Constraints . . . 86
4.8.1 Design Constraints . . . 86
4.8.2 Practical Constraints . . . 88
4.8.3 Security Specific Constraints . . . 89
4.8.4 Verification and Validation Constraints . . . 90
4.9 Industry Constraints . . . 91
4.9.1 Economic Constraints . . . 91

5 Application of ChatGPT and Similar Platforms . . . 92
5.1 Example 1 - Sourcing RTL files for the small configuration of the Ibex Core . . . 92
5.2 Example 2 - LowRISC Toolchain Setup (DV & Co-simulation) . . . 93
5.3 Example 3 - Generating RTL hierarchy order for Synthesis Scripts . . . 94
5.4 Example 4 - EDA Tool Comparison for an ASIC Design . . . 96

6 Hardware & Software Design . . . 97
6.1 Core Architecture and Top-Level . . . 97
6.1.1 ibex_top.sv . . . 98
6.1.2 ibex_top_tracing.sv . . . 98
6.1.3 ibex_core.f . . . 99
6.1.4 ibex_core.sv . . . 100
6.1.5 ibex_pkg.sv . . . 100
6.2 Pipeline Stages and Datapath . . . 102
6.2.1 ibex_alu.sv . . . 103
6.2.2 ibex_ex_block.sv . . . 104
6.2.3 ibex_id_stage.sv . . . 104
6.2.4 ibex_if_stage.sv . . . 105
6.2.5 ibex_wb_stage.sv . . . 105
6.2.6 ibex_multdiv_fast.sv . . . 105
6.2.7 ibex_multdiv_slow.sv . . . 106
6.3 Control and Decode Logic . . . 107
6.3.1 ibex_branch_predict.sv . . . 108
6.3.2 ibex_compressed_decoder.sv . . . 108
6.3.3 ibex_controller.sv . . . 109
6.3.4 ibex_decoder.sv . . . 109
6.3.5 ibex_dummy_instr.sv . . . 110
6.4 Memory System and Bus Logic . . . 110
6.4.1 ibex_fetch_fifo.sv . . . 111
6.4.2 ibex_icache.sv . . . 111
6.4.3 ibex_load_store_unit.sv . . . 111
6.4.4 ibex_pmp.sv . . . 112
6.4.5 ibex_prefetch_buffer.sv . . . 112
6.5 System and Debug Architecture . . . 113
6.5.1 ibex_counter.sv . . . 113
6.5.2 ibex_cs_registers.sv . . . 114
6.5.3 ibex_csr.sv . . . 114
6.5.4 ibex_lockstep.sv . . . 115
6.5.5 ibex_register_file_ff.sv . . . 115
6.5.6 ibex_register_file_fpga.sv . . . 115
6.5.7 ibex_register_file_latch.sv . . . 115
6.5.8 ibex_tracer.sv . . . 116
6.5.9 ibex_tracer_pkg.sv . . . 116
6.6 Core Redesign for Security . . . 116
6.6.1 Overview . . . 116
6.6.2 Boolean Masking to Mitigate First Order Attacks . . . 118
6.6.3 Random Noise Injection Implementation . . . 120
6.6.4 Cellular Automata Shift Register . . . 122
6.6.5 Clock Generator and Randomizer . . . 125
6.7 Verification Architecture . . . 128
6.7.1 Objectives and Approach . . . 128

6.7.2 Verification of Custom Sub-modules . . . 129
6.8 Ibex UVM Testbench Architecture . . . 134
6.9 Realization of the DV and Co-Simulation Environment . . . 139
6.9.1 Using the Ibex DV and Co-Simulation Environment in Practice . . . 141
6.10 Post-Synthesis Verification . . . 144
6.10.1 Masking after Applied Synthesis . . . 144
6.10.2 Execution Stimulus Model (C to VMEM Flow) . . . 144
6.10.3 Functional Coverage and Unreachability Analysis . . . 145
6.11 Physical Design . . . 145
6.11.1 Simulation: From Design Specification to Gate-Level Behavior . . . 146
6.11.2 Synthesis: Logic and Design-for-Test (DFT) . . . 147
6.11.3 Automatic Test Pattern Generation (ATPG) . . . 147
6.11.4 Physical Implementation . . . 147
6.11.5 Static Timing Analysis (STA) . . . 151
6.11.6 Logic Equivalence Checking (LEC) Across Synthesis and Implementation 151

7 System Testing and Evaluation . . . 153
7.1 Use of Ibex DV and Co-Simulation for Pre-Synthesis System Testing . 153
7.1.1 Regression Strategy and Coverage . . . 153
7.1.2 Evaluation of DV/Co-sim Results . . . 154
7.1.3 GLS Functional Coverage Evaluation and Reporting . . . 155
7.2 Validation of Post-RTL Security Integrity . . . 157
7.3 Security Sign-off Verification . . . 157
7.3.1 Timing Checks and Data Control . . . 157
7.4 Comparative SCA Methodology . . . 157
7.4.1 Baseline Characterization and Attacker Model . . . 158
7.4.2 Leakage Models and Technical Rationale . . . 159
7.4.3 Statistical Procedures and Controls . . . 160
7.5 Physical Validation Plan (Stretch Goal) . . . 161
7.5.1 FPGA Implementation . . . 161
7.5.2 Main Experimental Controls . . . 162
7.5.3 Expected Outcomes . . . 162

8 Administrative Content . . . 163
8.1 Bill of Materials . . . 163
8.2 Budget and Financing . . . 164
8.3 Project Milestones . . . 164
8.3.1 Senior Design 1 . . . 164
8.3.2 Senior Design 2 . . . 166
8.3.3 Distribution of Work . . . 166
8.3.4 Project Milestone Summary . . . 167

9 Conclusion . . . 169

A Software Code . . . 177

A.1 Module Designs (Verilog) . . . 177
A.1.1 casr37.v . . . 177
A.1.2 clkgen.v . . . 178
A.1.3 lfsr8.v . . . 179
A.1.4 lfsr43.v . . . 180
A.1.5 counter_5bit.v . . . 181
A.1.6 ring_oscillator.v . . . 181
A.2 Testbench Designs (Verilog) . . . 182
A.2.1 `casr37_tb.v` . . . 182
A.2.2 `tb_clkgen.v` . . . 184
A.2.3 `lfsr8_tb.v` . . . 185
A.2.4 `tb_lfsr43.v` . . . 187

B GLS Program Sources (C) . . . 189
B.0.1 `Shared Support: simple_system_common.c` . . . 189
B.0.2 `bubblesort_test.c` . . . 189
B.0.3 `crc32_test.c` . . . 190
B.0.4 `fibonacci_quick_demo_test.c` . . . 191
B.0.5 `fibonacci_test.c` . . . 192
B.0.6 `fibonacci_verbose_test.c` . . . 193
B.0.7 `gls_branch_trap_pipe_stress_test.c` . . . 194
B.0.8 `gls_mmio_be_cov_followup_test.c` . . . 196
B.0.9 `gls_mmio_be_cov_strong_test.c` . . . 198
B.0.10 `gls_mmio_be_cov_test.c` . . . 199
B.0.11 `gls_pipe_flow_cov_test.c` . . . 200
B.0.12 `gls_quick_cov_test.c` . . . 201
B.0.13 `irq_misaligned_test.c` . . . 202
B.0.14 `minimal_timer_test.c` . . . 203
B.0.15 `NGDemo.c` . . . 204
B.0.16 `periph_read_test.c` . . . 204
B.0.17 `riscv_instr_test.c` . . . 205
B.0.18 `sieve_test.c` . . . 207
B.0.19 `timer_irq_mmio_test.c` . . . 208
B.0.20 `timer_test.c` . . . 209
B.0.21 `UCFDemo.c` . . . 210

C ChatGPT Prompts and Outcomes . . . 212
C.1 Case Study 1: Can You Cause Jitter with an RTL Version of a Ring Oscillator? . . . 212
C.2 Case Study 2: Ibex DV Setup Chat Transcript . . . 213
C.3 Case Study 3: Generating RTL-file hierarchy order for Synthesis Scripts 229
C.4 Case Study 4: EDA Tool Comparison for an ASIC Design . . . 232
C.4.1 Exchange 1 . . . 232
C.4.2 Exchange 2 . . . 234
C.4.3 Exchange 3 . . . 235

List of Tables

2.1 Engineering Specifications 15

3.1 RISC-V Core Feature Comparison Table 30
3.2 EDA Tool Comparison 32
3.4 Comparison of Verification Strategies 34
3.5 Overview of Random and Pseudo-random Number Generators 36
3.6 Comparison of Masking Techniques for Side-Channel Attack Resistance 43
3.8 Comparison of Hiding Techniques for Side-Channel Resistance 48
3.9 Comparison of Open-Source and Generic PDKs 52
3.10 Build/SoC Framework Comparison 65

4.1 Summary of Relevant Cryptographic Standards 70
4.2 Comparison of Relevant Design Standards 72

6.1 Definitions Made in ibex_pkg 101
6.2 Signals in ring_oscillator Module 121
6.3 Signals in `casr37` Module 124
6.4 Table of Signals in Clock Generator and Randomizer Module 126
6.5 Table of signals for Perturbation Logic 128
6.6 Observed behavior of key signals in the `casr37` waveform. 130
6.7 Observed behavior of key signals in the `lfsr43` waveform 132

7.1 Interrupt Latency Measurement Results (DV/Co-sim, 9 Seeds, 200 Interrupts/Seed) 155
7.2 GLS Functional Coverage Closure Summary (Merged IMC Dataset) . 156
7.3 Side-Channel Leakage Model Comparison 160

8.1 Bill of Materials (BOM) 163
8.3 Senior Design 1 Timeline 165
8.4 Senior Design 2 Timeline 166
8.6 Work Distribution 166

List of Figures

2.1 Ibex Core diagram with color-coded assignments 17
2.2 ASIC Design Flow Process 18

4.1 The RISC-V Integer Register Set (x0–x31) 74
4.2 Instruction Types for RV32I ISA 74
4.3 The RISC-V Floating-Point Register Set (f0–f31) 76

6.1 Ibex Top Module 98
6.2 Ibex Top Tracing Module 99
6.3 Ibex Pipeline Stages 103
6.4 5-bit Baugh-Wooley Multiplication 107
6.5 Diagram of Ibex Core [1] 116
6.6 Schematic of Redesign of Ibex Core 117
6.7 Masking versions for the non-linear gates 118
6.8 Core schematic 119
6.9 Top Module Architecture 119
6.10 Schematic of Ring Oscillator 120
6.11 Schematic of 37-bit CASR 122
6.12 Schematic of Clock Generator and Randomizer 125
6.13 Schematic of Perturbation Module 127
6.14 `casr37.v` testbench waveform showing reset, enable gating, and CASR evolution with and without perturbations. 129
6.15 lfsr43.v Testbench Waveform 131
6.16 lfsr43.v Testbench Waveform (zoomed out) 131
6.17 Verification of *o_ptb* through XOR-reduction of the LFSR bits *o_state*[27:21]. Each row corresponds to one clock cycle from the waveform. The computed XOR value matches the RTL output *o_ptb* for all cycles, confirming correct implementation of the expression $o\_ptb = {}^{\wedge}o\_state[27:21]$. 132
6.18 `lfsr8.v` testbench waveform showing reset, pure LFSR operation, and `i_ptb` mixing over an extended run. 133
6.19 Example LFSR phase when `i_ptb_valid = 0` 134
6.20 Ibex UVM Testbench Architecture Overview 136
6.21 RISC-V-DV Test Generation and Integration Flow 137
6.22 Co-Simulation Software Stack for Ibex and Spike 138
6.23 Per-Instruction Co-Simulation Sequence Between Ibex and Spike . . . 139
6.24 Verification of the local Python virtual environment on `vlsi1`. 140
6.25 Sample simvision waveform of `riscv_machine_mode_rand_test` 142

6.26 Sample `regr.log` output of `riscv_machine_mode_rand_test` 142
6.27 sample `trace_core.log` output of `riscv_machine_mode_rand_test` . 143
6.28 sample `spike_cosim_trace_core.log` output of `riscv_machine_mode_rand_test` 143
6.29 Realization of RTL-to-GDSII Flow. Outlines the approximate, generalized workflow from start-to-finish for realizing a GDSII file output. . . 146
6.30 Final Physical Design in Cadence Innovous. 149
6.31 Timing analysis results with negative slack violation of $-1.035ns$ . . . 150
6.32 Final post-route timing results with 0 violations across all paths . . . 150
6.33 Design meeting timing after STA 151

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. NG RAD aims to address this gap by creating an extension for a RISC-V processor that can mitigate side-channel attacks through effective leakage reduction techniques. Our proposal frames the effort by gaining familiarity with ASIC design processes and utilizing Cadence EDA tools to develop side-channel-aware RTL into an ASIC design suitable for fabrication.

NG RAD 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 intended to mitigate Side-Channel Attacks into a physical chip implementation, demonstrating that the design is hardware-ready and suitable for fabrication. Given that one of the main objectives of this project is to gain familiarity with ASIC design workflows, it is essential to utilize industry-standard EDA tools such as the Cadence Design Suite.

Cadence provides an automated flow that streamlines the conversion of behavioral-level circuit descriptions into a GDSII format, which is ultimately used to fabricate the circuit on silicon. This project will require the use of multiple tools within the Cadence suite, including Genus, Xcelium, SimVision, and Innovus. These tools automate major steps in the design process, operating significantly faster and more efficiently than manual gate-level design. Cadence Xcelium is used to simulate the RTL design alongside its testbench, enabling debugging and ensuring that the intended functionality is correctly expressed. Once waveforms are generated through simulation, they can be observed in SimVision, Cadence's waveform viewer. In introductory digital logic classes, testbenches are created for small designs where all inputs and states can be exhaustively tested, but as designs increase in complexity, as is the case for this project, simple functional verification becomes insufficient. Therefore, the team will verify the design using the Universal Verification Methodology (UVM), a standardized verification framework for validating digital logic and ensuring that the design behaves according to specification. Within the context of Side-Channel Analysis Attacks, these verification techniques allow us to confirm with confidence that the RTL implementations of mitigation algorithms are functionally correct.

After verification, the design is passed to Cadence Genus, which synthesizes the RTL into a netlist describing the logical components of the circuit. Synthesis proceeds in three main steps: syn_generic, syn_map, and syn_opt. These steps respectively generate the initial netlist, map that netlist onto the standard cell library provided by the Process Design Kit (PDK), and optimize the result to meet timing and area constraints. In the context of Side-Channel Analysis Attacks, synthesis ensures that the RTL can be correctly transformed into an equivalent gate-level representation. If synthesis fails, such as when the RTL contains unsynthesizable constructs, the flow

logs will report errors, indicating implementation issues that must be addressed.

Once synthesis is complete, the design is taken into Innovus for physical implementation. Here, the team will floorplan the chip (define die size, pin placement, and macro locations), place the gates, and perform routing. Innovus automates most of the place-and-route process but still requires user guidance to validate results and manually resolve any design rule violations. These violations may provide insight into layout-level issues that could affect the correctness or robustness of our Side-Channel Attack mitigations.

Overall, Cadence tools are essential for developing a microarchitecture that incorporates Side-Channel Attack mitigation techniques. The suite enables thorough verification of RTL functionality, synthesis into a correct gate-level design while detecting unsynthesizable code, and successful placement and routing to prepare the design for tapeout. Without these tools, traditional methods such as manual gate-level design and layout would be far more error-prone, inefficient, and impractical for a project of this scale.

Furthermore, the Cadence flow allows designers to estimate dynamic and static power during both synthesis and place-and-route. These power estimates provide early insights into whether additional balancing or masking logic is required to reduce potential leakage sources before fabrication. Incorporating these evaluations into the standard EDA flow ensures that security-oriented design choices remain consistent throughout the entire development pipeline.

2.1.2 Importance of Hardware Security

With the increasing presence of IoT and connectivity in modern electronics, significantly more resources have been allocated to securing our devices from malicious entities. As a result, the cybersecurity community has made impressive advancements in software-based protections, yielding substantial results. Although this progress is valuable, far more emphasis must be placed on defending against hardware-focused attacks, since electronic systems are equally, if not more, susceptible to physical compromise. These attacks are particularly dangerous because once the hardware is breached, the entire system becomes vulnerable or even unusable. An attacker who successfully compromises the physical components is often able to bypass software security mechanisms entirely.

Successful hardware attacks can result in stolen intellectual property, identity theft, exposure of sensitive customer data, and the subversion of critical embedded systems. The stakes become even higher when considering applications within the Department of Defense (DoD), where devices are responsible for mission-critical operations, can cost tens of millions of dollars, and may directly impact human safety. Defending these systems is especially challenging due to their complexity, long deployment lifetimes, and the diverse environments in which they operate. This creates a scenario where large teams of skilled engineers are required to ensure robust protection, while adversaries often have abundant time, funding, and advanced techniques to identify and exploit even the smallest design weakness.

The problem is further amplified by the inherent imbalance between attackers

and defenders. The attacker only needs to discover one vulnerability to cause significant damage, whereas the defender must anticipate and mitigate all possible attack surfaces. Once a hardware system is compromised, the consequences can be catastrophic, particularly because detecting information leakage or addressing underlying design flaws after deployment is extremely difficult. These realities emphasize the critical need for hardware-level security research, rigorous verification, and thoughtful architectural design to protect the next generation of electronic systems.

2.1.3 Types of Hardware Attacks

Unlike software attackers, who 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 among the most prevalent and dangerous forms of hardware compromise, primarily because they exploit unintended information leakage rather than structural vulnerabilities in the design itself [3]. These leakages arise from a chip's normal physical behavior, including its power consumption, temperature fluctuations, electromagnetic emissions, or even the timing differences between executed instructions. Since these signals are naturally produced by standard operation, defending against their misuse is particularly challenging. One widely used class of side-channel exploits involves time-driven attacks, which allow an adversary to infer a device's control flow and internal operations by measuring how long instructions take to execute. For example, by observing the latency of instructions following a branch, an attacker can deduce whether the processor followed a long or short execution path. Similarly, by timing memory operations, an attacker can determine whether a cache access resulted in a hit or a miss, revealing which data or code paths were accessed.

Beyond timing channels, attackers can examine power-related leakage to recover sensitive information. Simple Power Analysis (SPA) directly inspects instantaneous power consumption to interpret device behavior. Despite being a comparatively low-

effort attack, SPA can still leak cryptographic keys or reveal whether an arithmetic instruction is performing addition or multiplication. More advanced attacks such as Differential Power Analysis (DPA) collect many power traces and statistically correlate them to extract subtle patterns hidden beneath noise. Although DPA is computationally more expensive, it is significantly more effective because its statistical techniques can overcome randomization, noise injection, and other first-order defenses. While SPA typically exploits straightforward leakage from unprotected implementations, DPA systematically defeats more sophisticated countermeasures by treating them as delays rather than genuine sources of entropy.

Although traditional side-channel attacks are considered passive (where the adversary only observes unintended leakage produced during normal execution) modern microarchitectural exploits blur this distinction. Attacks like Spectre and Meltdown induce transient or speculative behaviors to force the microarchitecture into revealing secret-dependent information through timing or cache-based channels. This means that even though the measurements themselves are passive, the setup phase is active, placing many real-world exploits in a hybrid category: they actively manipulate execution paths and then passively measure the resulting physical or microarchitectural leakage.

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 [4], 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 [5], 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 [6].

Subsequent work has turned RowHammer into more practical exploits that impact web browsers, mobile applications, and perhaps most credibly cloud computing [7]. 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 [5]. 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 [8].

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 PRNG 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 [5]. 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 [9] [10]. 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 [11], 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 [12]. 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 rising prevalence of side-channel attacks has motivated the development of new mitigation techniques and processor architectures aimed at reducing their impact. This section reviews prior research and commercially deployed solutions that incorporate hardware-level features to strengthen processor security.

Understanding existing approaches to side-channel defense is essential for identifying effective strategies and informing architectural decisions. By examining previous work, we establish the context for the design choices made in this project and highlight the techniques that guided our implementation.

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

In the study by Mulder et al. [13], the authors present a RISC-V processor architecture specifically designed to mitigate vulnerabilities that arise during cryptographic operations, both within the Arithmetic Logic Unit (ALU) and during memory

accesses involving secret data. Their work aims to reduce the burden placed on software developers by shifting much of the security responsibility into hardware. Rather than requiring extensive secure coding practices, the processor incorporates built-in countermeasures that automatically protect sensitive operations. Central to their approach is the use of masking techniques that ensure any data being processed or transferred remains statistically independent of the underlying secret values.

For ALU operations, Mulder et al. adopt Boolean masking based on the d + 1 Threshold Implementation (TI) methodology. This technique introduces randomness by combining sensitive data with a mask, producing "shares" of the original value. To achieve first-order power-analysis resistance, the data are split into two independent shares, each processed separately through the ALU pipeline. Intermediate results never directly expose the secret value and are only recombined once computation completes. By distributing computation across these two shares, the design prevents data-dependent power variations and mitigates leakage caused by transient hardware effects such as signal glitches. This structured masking approach ensures that the ALU's power consumption patterns remain uncorrelated with the actual secret values being manipulated.

Memory operations required a different strategy. Storing both the masked data and its corresponding mask in memory would introduce a significant vulnerability, especially when both values travel across the same data bus. To avoid this, the authors developed a memory-independent masking mechanism. Their solution generates a second mask deterministically using the memory address and several predefined seeds chosen by the software developer. This allows the system to reconstruct masked values during encryption and decryption operations without ever storing the mask itself. Additionally, the design supports explicit software-triggered reseeding, which ensures that sensitive masked values become invalidated once they are no longer needed, preventing potential reuse-based attacks.

Experimental evaluation demonstrated that these methods substantially suppressed power-based leakages across the processor. Despite the promising results, the authors note challenges related to the processor's closed-source nature, which limits transparency into its internal implementation details. Furthermore, delegating certain tasks, such as reseeding control, to secure software components can introduce complexity in validation and testing workflows. Even with these limitations, the Boolean masking strategy proved highly effective: across more than two million power trace measurements, no meaningful information leakage was detected except during brief phases at the start and end of execution, where typical initialization effects occur. This work highlights the practical strength of hardware-level masking techniques and provides a strong foundation for future secure processor designs.

2.2.2 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.

2.2.3 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.

2.3 Goals and Objectives

Currently, 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 FPGA 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	0.9 V – 1.2 V	High
Process Node	∼45 nm	High
IR Drop	$\leq 5\%$	High
Average Frequency	≥ 15 MHz	High
Masking Order	First-order minimum	High
Chip Density	$\geq 60\%$	Medium
Cache Size	≥ 4 kB	High
Timing Failure Count	0 Faults	High
Interrupt Latency	≤ 12	Medium
Leakage Power (25 °C)	$\leq 5mW$	High

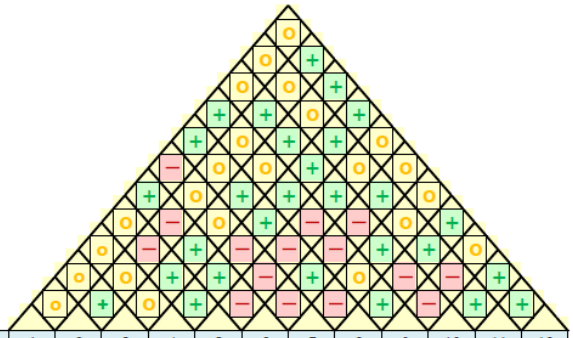
2.5 House of Quality

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	Chip Density	Cache Size	Timing Failure Count	Interrupt Latency	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	0.9V - 1.2V	~ 45nm	≤ 5%	≥ 15 MHz	First-order minimum	≥ 60%	≥ 4 kB	0 Faults	≤ 12 cycles	≤ 5mW

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 this project'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:

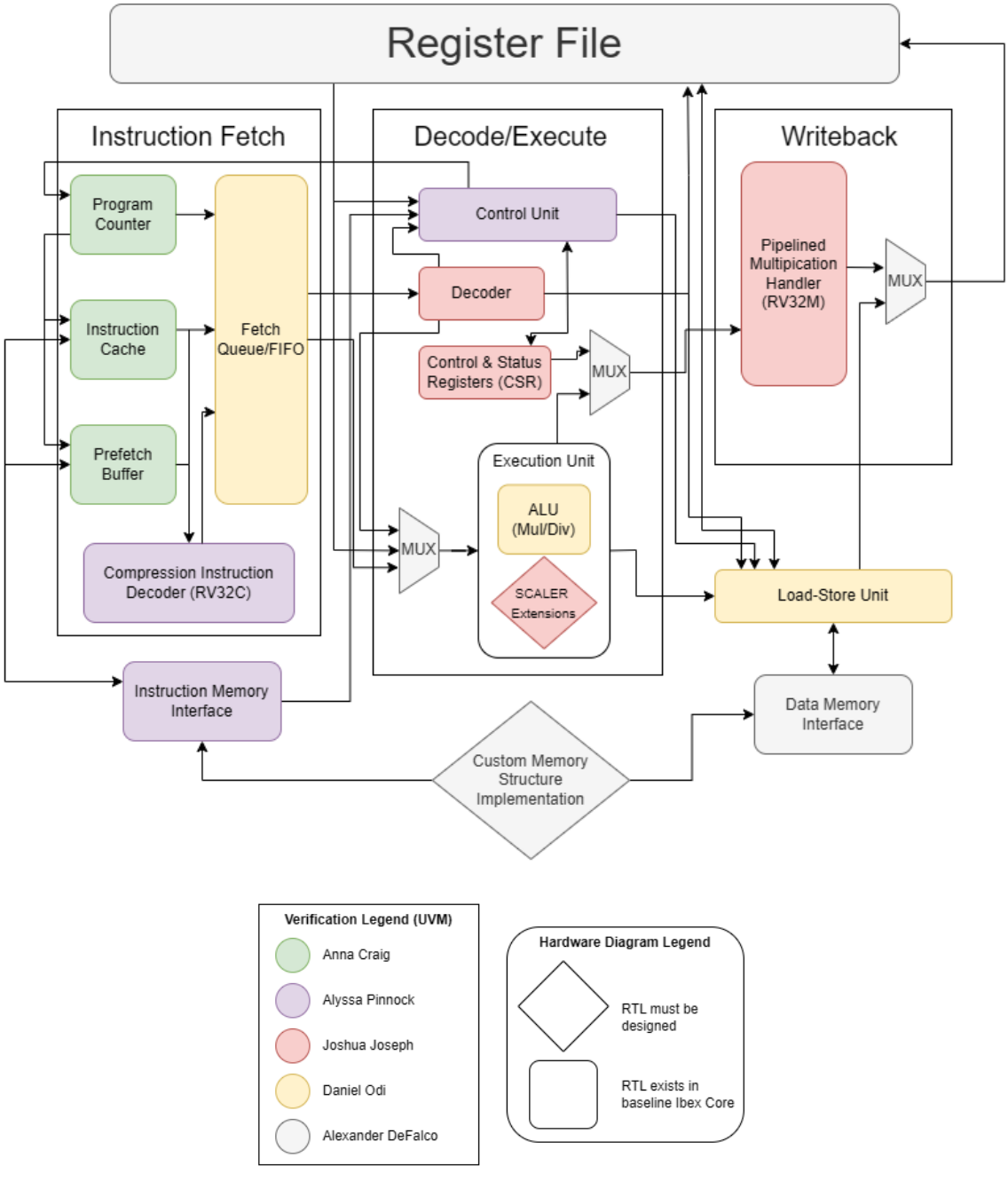


Figure 2.1: *Ibex Core diagram with color-coded assignments*

It is worth recognizing that the added 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 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.

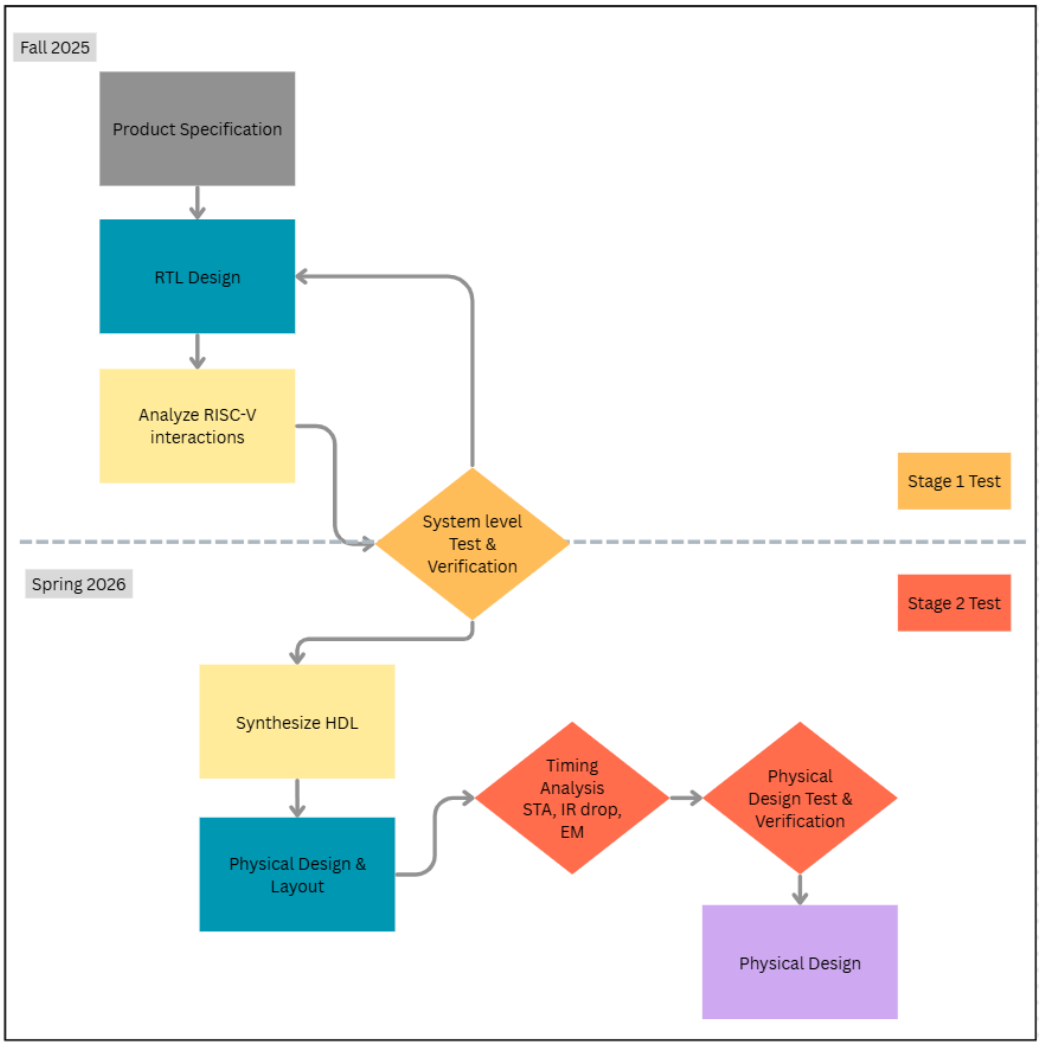


Figure 2.2: *ASIC Design Flow Process*

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 in the process will work together to meet the required specifications.

Core functional requirements for the ASIC design are defined in the specification document. These functional requirements include descriptions of 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 must also be addressed at multiple levels of implementation to ensure the design meets its power constraints.

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. The floorplan serves as the physical layout that placement and routing will build upon 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 are 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. Verilog's 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

HDLs 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.

HDLs 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. Verilog's 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 [4]). 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 [1].

Ibex focuses heavily on industry-grade verification. The design has been formally verified with open-source frameworks, and the UVM environments, assertions, 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 [23] [24]. 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, trading 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: *RISC-V Core Feature Comparison Table*

Core	Micro-architecture	Language	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 modifications
NEORV32	In-order, 32-bit MCU-class	VHDL	Good self-tests/examples emphasis on precise exceptions	Beginner-friendly; safety/predicta 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-architectural testing	High performance; multi-level caches and MMU; vector support	Very complex; heavy dependencies; partial English documentation; 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 environment; formal proofs; OpenTitan DV	Clear SV RTL; PMP support; flexible configs; security-minded design	No MMU/caches; modest performance; not Linux-class

3.3 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.

3.3.1 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.

3.3.2 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].

3.3.3 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.

3.3.4 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 3.2: *EDA Tool Comparison*

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.

3.4 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.

3.5 Verification Methodology Selection

A key early design decision for this project was the choice of verification methodology for the modified Ibex core. Rather than developing a new processor-level testbench from the ground up, we opted to reuse and extend the existing Ibex design verification (DV) and co-simulation infrastructure maintained by lowRISC. The Ibex core ships with a SystemVerilog UVM environment as well as a lockstep (synchronized) co-simulation system which compares the RTL against a reference Instruction Set Simulator (ISS). Randomized instruction programs are generated using the RISC-V-DV framework, compiled, and then executed on both the Ibex core and a customized fork of the Spike RISC-V ISS. During simulation, all architecturally visible events (retired instructions, write-backs, memory operations, etc.) are continuously checked against the ISS to detect mismatches and discrepancies.

We considered three broad options for processor and SoC verification in this project. The first option was to rely on directed tests with simple self-checking testbenches. This approach offers a very simple initial setup and is easy to understand, but it provides poor coverage for corner cases and is weak at catching subtle bugs in the microarchitecture. The second option was random instruction testing without co-simulation, in which RISC-V-DV generates randomized instruction sequences and we only check final pass/fail outcomes. This yields higher coverage than purely directed tests, but failures are harder to debug because only end-of-test values are available and intermediate behavior is not directly observed. The third option was random instruction testing with lockstep co-simulation, leveraging the existing Ibex DV environment and the lowRISC Spike fork. This flow provides strong architectural checking, detailed visibility into each retired instruction, and benefits from an actively maintained infrastructure with a community for support, at the cost of a more complex build flow and simulation setup that can require additional effort during initial integration.

Table 3.4: *Comparison of Verification Strategies*

Approach	Pros	Cons
Directed tests with simple self-checking testbenches	Simple initial setup; easy to understand and reason about.	Poor coverage for corner cases; weak at catching subtle bugs in the microarchitecture.
Random instruction testing without co-simulation (RISC-V-DV sequences, pass/fail only)	Higher coverage than purely directed tests; able to explore a wider variety of instruction mixes and scenarios.	Harder to debug failures; only end-of-test values are visible, with no detailed insight into intermediate behavior.
Random instruction testing with lockstep co-simulation (Ibex DV + Spike fork)	Strong architectural checking; detailed visibility into each instruction; leverages an existing, actively maintained infrastructure with a community for support.	More complex build flow and simulation setup; may involve additional overhead and complexity when initially building and integrating tests.

After much discussion and analysis of these trade-offs, the third option was opted for: reusing the existing Ibex UVM + co-simulation stack as our baseline verification methodology. This choice gives us a few luxuries:

1. We can inherit a mature, coverage-oriented verification flow that has already been proven through OpenTitan configurations of Ibex (Earl Grey).

2. Leverage RISC-V-DV to stress both the baseline core and our extensions with randomized programs.

3. Leverage the Spike co-simulation as a reference to precisely identify regression bugs introduced by our core modifications.

For the purpose of research, it is sufficient to view the Ibex DV environment as a pre-existing technology that has been evaluated and selected. More detailed architectural overviews and modification plans for the testbench/co-simulation system(s) will be discussed further in Chapters 6 and 7.

3.6 Randomness vs. Pseudo-randomness

In many of the hiding and masking techniques used throughout this work, the word "random" is used informally to describe values that are unpredictable to an adversary. In practice, it is important to distinguish between true randomness, which

is derived from physical phenomena, and pseudo-randomness, which is generated deterministically by an algorithm from a short, secret seed. This distinction drives how randomness is produced in hardware and how robust a given countermeasure is against side-channel and fault attacks.

A truly random source is modeled as a physical process whose outputs have high entropy and cannot be predicted better than by chance, even if an attacker knows the entire design of the system. In hardware, this is typically realized by a true random number generator (TRNG) that samples analog noise: thermal noise in transistors, clock jitter in ring oscillators, or metastability in latches and arbiters. Small, uncontrollable variations in delay and voltage introduce timing jitter and metastable behavior, and when these are digitized they produce bit streams that, under suitable conditions, can be treated as true random outputs. Standards such as NIST SP 800-90B formalize this notion by defining an "entropy source," specifying how to assess its min-entropy and how to apply health tests so that gross failures (for example, stuck oscillators) can be detected [28].

In contrast, digital computers are fundamentally deterministic devices: given the same initial state, they always produce the same outputs. As a result, most "random" values produced in software (for example, by rand(), /dev/urandom, or cryptographic libraries) are generated by pseudo-random number generators (PRNGs). A PRNG takes a short seed and expands it into a much longer sequence of bits that appears random to any efficient adversary who does not know the seed. Cryptographic PRNGs, also called deterministic random bit generators (DRBGs) in NIST terminology, are typically built from standard primitives such as hash functions, HMAC, or block ciphers in counter mode, and are specified in NIST SP 800-90A [29]. From a modern cryptographic viewpoint, a well-designed DRBG produces outputs that are computationally indistinguishable from a truly random sequence as long as the seed has sufficient entropy and remains secret [30].

Because of this, modern systems commonly combine both worlds: a noise-based TRNG periodically samples physical entropy and uses it to seed (and occasionally reseed) a deterministic generator, which then provides high-throughput pseudo-random bits for masks, nonces, and other protocol values [29, 28]. NIST SP 800-22 defines statistical tests that can be applied to both true and pseudo-random generators to detect obvious patterns or biases in their output, but passing these tests alone does not guarantee cryptographic security [31].

Following sections focus on practical ways to implement both types (random and pseudo-random) of generators in hardware and evaluate their suitability for side-channel countermeasures/mitigation techniques.

Table 3.5: *Overview of Random and Pseudo-random Number Generators*

Type	Source / Mechanism	Typical Role / Notes
True Random Number Generator (TRNG)	Physical noise (thermal noise, jitter, metastability)	High-entropy, unpredictable bits; used to seed other generators and generate long-term keys.
Non-cryptographic PRNG (e.g., `rand()`)	Simple deterministic algorithm	Fast and easy to implement; suitable for simulations and non-security uses; not safe against attackers.
Cryptographic PRNG / DRBG	Deterministic generator built from hash, HMAC, or block cipher	High-throughput "random-looking" bits from a secret seed; suitable for masks, nonces, and protocol values.
Hybrid TRNG + DRBG	TRNG seeds and occasionally reseeds a DRBG	Common system design; combines true entropy with efficient deterministic expansion.

3.6.1 Linear Feedback Shift Registers (LFSRs)

Linear Feedback Shift Registers are pseudo-random number generators (PRNGs) that are attractive because of their low hardware overhead. LFSRs consist of a shift register where 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[32]. If the XOR taps are set for the maximum period, for an 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 (PRNGs). The period of the LFSR, the amount of numbers generated before returning 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 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 is 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 predictability as a result of their pseudo-random nature leads them to be vulnerable to side-channel analysis techniques. By themselves, LFSRs are not attractive as something to implement in this project, but in conjunction with other techniques, they prove to be promising, specifically when introducing entropy sources into their number generation to increase randomness.

3.6.2 Ring-Oscillators

ring oscillator is a fundamental electronic circuit in which the output of the last inverter is fed back into the input of the first, forming a closed feedback loop. This configuration causes the circuit to oscillate, producing a periodic waveform. The oscillation frequency depends on the propagation delay of each inverter stage, which is influenced by physical factors such as temperature, supply voltage, and process variations[33]. These factors introduce small, random fluctuations in the oscillation period, making the frequency inherently non-deterministic.

Jitter refers to these small, random variations in the timing of oscillations. In a ring oscillator, jitter is primarily caused by thermal noise, supply voltage noise, and intrinsic device-level variations. By sampling the phase of the ring oscillator at very short intervals, the resulting variations can be recorded as binary values. These fluctuations serve as a source of entropy, and the randomness of the jitter makes it difficult for an attacker to predict future samples or reconstruct the underlying process.

Ring oscillators differ from LFSRs and white-noise generators in that they can produce true randomness directly from physical phenomena. When properly designed, they can generate entropy without the need for complex post-processing or specialized hardware. However, ring oscillators do not inherently generate usable random numbers; instead, the physical randomness they produce is often used to seed or enhance pseudo-random number generators, improving the quality of their output.

Despite their potential, ring oscillators have a key limitation in digital design flows. Clock jitter and other forms of physical randomness cannot be fully realized through RTL synthesis alone. RTL synthesis converts high-level designs into a fixed gate-level netlist, which is inherently deterministic given specific inputs and clocking. As a result, any random behavior implemented purely in RTL is pseudo-random, relying on algorithmic methods rather than true physical entropy. Without incorporating analog components or other physical entropy sources, RTL-to-GDSII flows cannot generate genuine stochastic behavior. Given that the focus of this project is the RTL-to-GDSII flow, this motivates the use of pseudo-random number generators (PRNGs) that approximate the behavior of true random number generators (TRNGs) rather than attempting to implement a TRNG directly.

3.6.3 Gaussian Number Generators

Gaussian noise generators are fundamental digital signal processing components that produce random samples following a Gaussian (normal) probability distribution, serving as essential tools in modern communication systems and measurement equipment. These generators create white Gaussian noise with standard normal distribution characteristics, enabling engineers to accurately model and test system performance under realistic channel conditions. The importance of high-quality Gaussian noise generation is particularly evident in applications requiring bit error rate measurements down to extremely low levels such as 10^{-15}, where the accuracy and statistical properties of the generated noise become critical. The core challenge in designing these generators lies in transforming uniformly distributed random variables into properly distributed Gaussian samples with sufficient range and precision. Several algorithmic approaches exist for this transformation, each with distinct trade-offs between complexity, throughput, and accuracy. The Central Limit Theorem approach [34], while conceptually simple, requires summing many samples and proves unsuitable for high-speed applications. The popular Box-Muller method relies on transcendental functions like square root, logarithm, and trigonometric operations, which can be resource-intensive in hardware. Rejection-acceptance methods such as the Polar and Ziggurat algorithms, though efficient in software, suffer from nonconstant output rates due to their conditional branching structure, making them problematic for hardware implementations requiring deterministic timing. Modern hardware implementations increasingly favor the inversion method, which applies the inverse cumulative distribution function to transform uniform random inputs into Gaussian outputs, enabling generation of random samples for arbitrary distributions. This approach supports high throughput rates exceeding 300 MHz in advanced FPGA implementations. A key performance metric is the effective range of the distribution, with sophisticated designs achieving Gaussian characteristics extending to 9.1σ from the mean. The underlying uniform random number generator also significantly impacts overall quality, with combined Tausworthe generators offering superior randomness properties and extremely long periods. Advanced implementations can achieve periods of 2^{176} (approximately 10^{53}), ensuring negligible probability of sequence repetition during practical usage. Modern designs typically quantize output samples to 16 bits with 5 bits of integer and

11 bits of fraction, balancing precision against hardware resource requirements. The primary application domain encompasses communication systems requiring accurate emulation of additive white Gaussian noise channels and sophisticated bit error rate measurement systems. As wireless standards evolve and data rates increase, the demand for efficient, high-speed Gaussian noise generators continues to grow, making these IP cores valuable building blocks in contemporary digital communication test equipment and simulation platforms.

3.6.4 Statistical Testing Standards for PRNGs

Statistical testing is a core component for properly verifying any design, especially for Pseudo-Random Number Generators (PRNGs) in high-security applications that require random mask generation [35]. The main goal of these standards is to gather evidence that a generator can produce binary sequences that appear statistically indistinguishable from truly random data. The established benchmark for evaluating the statistical quality of a PRNG is the NIST 800-22 statistical test suite, as mentioned in Section 3.6: Randomness vs. Pseudo-randomness. Researchers often rely on this framework to determine if a PRNG is suitable for cryptographic applications. The NIST suite includes 15 distinct tests that are designed to detect various types of non-randomness with sequences [36].

The evaluation and testing of PRNGs requires a rigorous hypothesis testing framework. Each test applies a different statistical test to determine the null hypothesis that the sequence tested is random. The core metric to emerge from the statistical tests is the P-value, which describes the evidence against the null hypothesis [37]. It also represents the probability that a perfect random number generator would have produced a sequence that appears less random than the sequence that was originally tested. Interpretation strategies to be able to assess overall performance would involve analyzing the proportion of sequences passing a given test and checking the uniformity of the P-values across all sequences tested [35].

For designing an ASIC that focuses on hardware security and side-channel mitigation, statistical testing is essential so the verification team can properly validate that the generated inputs are statistically sound. Our project relies on many PRNGs, such as LSFRs, to create random masks the processor. The verification team can collect results from these generators for final evaluation based on the NIST standard. Passing these tests would further validate and confirm that the output possesses the required statistical quality to work in generating masks effectively.

3.7 Side-Channel Mitigation Techniques

The purpose of this project is to add features that are known to mitigate side-channel attacks on an existing processor. This section covers the current methods presented in hardware security research papers. The primary methods in the field are masking and hiding which will be detailed in later sections. Special considerations must be made to ensure the methods chosen do not overlap and are feasible given

constraints presented in the project.

Many of the mitigation techniques applied in this project stem from deliberate choices made during the engineering-specification phase. Parameters such as process node, operating-voltage range, IR-drop tolerance, timing-failure margins, operating frequency, IPC, operating-temperature range, and cache size all influence the degree of side-channel leakage present in a design. The rationale behind selecting these standards is discussed in Section 4.7 Engineering Standards Selection.

3.7.1 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 [38].

3.7.1.1 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 [39]. This method implemented on the SERV Core along with other masking and hiding techniques showed efficient mitigation of Correlation Power Analysis Attacks [24].

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 [24].

3.7.1.2 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.

3.7.1.3 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 boolean masking 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[40, 41]. Thus, other methods that detailed in later sections were explored and chosen as they were easier to integrate with Biruykov's

[39] optimal boolean masking techniques.

3.7.1.4 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 [42].

3.7.1.5 Domain-Oriented Masking

Domain-Oriented Masking aims to prevent leakage from hardware boolean-masking implementations due to 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 [42].

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 3.6: *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)
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)

3.7.1.6 Implementation of Masking Techniques

For each of these masking techniques, the preceding sections provide a high-level overview of their respective advantages and limitations. Another important consideration is how these techniques can be integrated into open-source RISC-V designs. Broadly, there are two primary implementation approaches.

The first approach involves modifying the RTL of existing RISC-V designs. In this method, 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, while linear operations like XOR can be applied independently to each share. This approach offers full design-time control and enables transparent functional verification. However, this level of control comes with significant design effort. Once the RTL has been adapted, additional challenges emerge during downstream EDA tool flows. During synthesis, EDA tools optimize the netlist to meet performance or area objectives, which may inadvertently compromise security. For example, an optimization that reduces gate count in a security-critical section may unintentionally introduce leakage vulnerabilities exploitable through side-channel analysis.

The second approach introduces masking post-synthesis [24]. In this method, masking is applied automatically to the synthesized netlist. Scripts identify logic cones driven by sensitive nets, duplicate them into share domains, and replace standard cells with masked cell templates from a custom library. This can be achieved by developing TCL scripts integrated into the synthesis process. Rather than redesigning the RTL, designers need only create the necessary secure custom cells, derived from the existing cell library, and incorporate the TCL automation. The main drawback of this gate-level transformation is reduced algorithmic transparency, which can obscure functional intent and makes formal equivalence checking essential.

3.7.1.7 Masking Technique Selection

First-order Boolean masking was chosen for the project because it provides an effective balance between security, implementability, and hardware efficiency while still benefiting from strong community support and existing open-source implementations. By splitting each sensitive value into two shares and ensuring all operations preserve this sharing, first-order masking protects against the most common single-point leakage attacks, such as Correlation Power Analysis, without introducing the substantial area and randomness overhead associated with higher-order or arithmetic masking schemes. Its compatibility with linear operations makes it straightforward to integrate into a processor data path, and recent work by Biryukov on optimal SecAND and SecOR gates further reduces randomness requirements by eliminating the need for fresh masks during nonlinear operations. This significantly lowers runtime cost and makes the technique more practical for a lightweight RISC-V core. Additionally, the approach benefits from prior open-source implementations that demonstrate how Boolean masking can be incorporated into RISC-V designs at the gate level using automated script-based transformations. These resources provide a proven foundation, reducing development effort and risk when adapting the technique to this project. Overall,

first-order Boolean masking offers an ideal combination of practicality, efficiency, and accessible reference designs, making it a natural choice for enhancing side-channel resistance in the core.

3.7.2 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.

3.7.2.1 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 independent 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.

3.7.2.2 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 [43]. 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[24].

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.

3.7.2.3 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 [44].

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.

Table 3.8: *Comparison of Hiding Techniques for Side-Channel Resistance*

Technique	Purpose / Mechanism	Advantages	Limitations / Challenges
Random Noise Injection	• Adds randomness to power usage • Uses LFSRs, ring oscillators, toggle logic • Lowers signal-to-noise ratio for attackers	• Simple hardware integration • Works with existing RTL/software • Complements other techniques	• Only hides leakage • Increases power overhead • May affect timing stability
Random Dynamic Frequency Scaling (RDFS)	• Randomizes operating frequency • Misaligns power traces • Uses PLLs/DCOs driven by random inputs	• Low hardware cost • Highly effective misalignment • Protects all workloads during use	• Still allows leakage with many traces • May cause timing or sync issues • Performance and power efficiency vary
Random Delay Instruction Insertion (RDII)	• Inserts random NOPs or stalls • Misaligns event timing • Added via compiler or hardware stalls	• Very low hardware cost • Adjustable randomness • ISA-compatible for RISC-V	• Requires toolchain modification • Adds execution time variability • Vulnerable to advanced alignment

3.7.2.4 Hiding Techniques Selection

Among the evaluated countermeasures, Random Noise Injection and Random Dynamic Frequency Scaling (RDFS) offer the most practical and impactful improvements to side-channel resistance without requiring invasive architectural or toolchain changes. Noise injection is particularly appealing because it can be implemented by integrating high-switching-activity circuits such as LFSRs, ring oscillators, or toggle logic directly into the processor. These modules can be connected through existing control signals, which keeps integration minimally disruptive while significantly reducing the signal-to-noise ratio available to an attacker. RDFS provides another strong layer of protection by introducing temporal randomness through a hardware clock randomizer that generates a varying clock for the processor, effectively misaligning power traces with negligible impact on RTL complexity. Both techniques operate entirely at the hardware level, align well with the project's goal of gaining experience with ASIC design, and avoid the substantial difficulty of modifying the RISC-V compiler toolchain. In contrast, Random Delay Instruction Insertion would require altering compiler internals and code-generation flows, which is impractical

given the specialized toolchain used for the Ibex core. Therefore, hardware-based noise injection and dynamic frequency scaling offer the best combination of practicality, effectiveness, and compatibility for this project.

3.7.3 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 [45].

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.

3.7.3.1 Skywater 130nm PDK

SkyWater's SKY130 is the flagship open-source PDK that enabled broad community access to manufacturable CMOS at a mature node. Delivered through a collaboration between SkyWater and Google, SKY130 bundles process design files, design rules, SPICE device models, LVS and DRC decks, and several foundry provided standard cell families such as the sky130_fd_sc_hd [46] family and related variants. It is packaged for popular open flows including OpenLANE and open_pdks and it benefits from extensive documentation, numerous test chips, and a highly active community that simplifies the path to first silicon for researchers, students, and small development teams. The main advantages include transparent licensing, strong interoperability with open-source EDA tools, a rich set of digital and IO cells suitable for synthesis, and a reasonable balance between analog friendliness and digital capability. This combination makes SKY130 attractive for mixed signal prototypes, educational tapeouts, and research projects that need a real and accessible fabrication process. The disadvantages primarily stem from the larger feature size. As a 130 nm process it cannot match the transistor density, switching speed, or power and performance characteristics of more advanced nodes such as 45 nm. Designs that target high performance, low power consumption, or minimal area will find SKY130 significantly larger and slower. When compared with GlobalFoundries GF180MCU, SKY130 offers a more active open-source ecosystem and a broader selection of open cell libraries, although GF180MCU provides better high voltage IO options and is often preferred for rugged microcontroller style applications. Against Cadence GPDK 45 nm, SKY130 gives up density, clock frequency, and area efficiency because GPDK 45 nm represents a more modern technology class, even though it is a generic and not a full production PDK. Overall SKY130 remains a leading choice for learning, prototyping, mixed signal experimentation, and accessible research, trading cutting edge metrics for openness and strong community support.

3.7.3.2 Global Foundries 130nm PDK

GlobalFoundries GF180MCU is an open-source version of a mature 180 nm microcontroller oriented process released through a collaboration between GlobalFoundries and Google in order to broaden access to a reliable and high voltage capable manufacturing technology. The GF180MCU PDK provides process rules, SPICE models, LVS and DRC decks, characterization data, and standard cell libraries that include both 7 track and 9 track options designed for MCU style digital logic [47]. The process supports 3.3 volt and 6 volt IO levels and places strong emphasis on analog and mixed signal functions, which makes it suitable for sensor interfaces, automotive peripherals, industrial control blocks, and other applications that demand wide voltage margins and electrical robustness. Strengths of GF180MCU include its well established analog device models, generous IO voltage tolerance, predictable matching behavior, and a conservative manufacturing profile that simplifies early stage verification.

These characteristics make it appealing for teams that prioritize reliability, analog performance, or integration with legacy components. The disadvantages stem from the relatively large 180 nm geometry, which leads to higher area, slower switching speeds, and higher dynamic energy per transition when compared to sub 100 nm processes. Designs that require compact layout, high clock frequencies, or aggressive power optimization will find GF180MCU significantly less competitive. When compared to SkyWater SKY130, GF180MCU often delivers better high voltage IO options and a process heritage that aligns with classic microcontroller design, while SKY130 usually offers more community support, more examples, and denser digital libraries. Against Cadence GPDK 45 nm, GF180MCU is outperformed in every scaling sensitive dimension. GPDK 45 nm provides much smaller transistors, shorter interconnects, and lower parasitics, all of which lead to higher performance, lower area, and lower energy per operation. However GF180MCU retains advantages in analog simplicity, wider voltage operation, and a friendlier environment for hand crafted layout. In practice GF180MCU occupies a stable niche where robustness, analog behavior, and IO flexibility are more important than density and speed. The selection between these PDKs should be based on voltage requirements, area constraints, desired clock frequencies, and available EDA tooling.

3.7.3.3 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 [48]. 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.

Table 3.9: *Comparison of Open-Source and Generic PDKs*

Feature	SkyWater SKY130 (130 nm)	GlobalFoundries GF180MCU (180 nm)	Cadence GPDK 45 nm
Process Node	130 nm CMOS	180 nm CMOS/MCU	45 nm CMOS (generic)
Standard Cell Libraries	Multiple families (e.g., `sky130_fd_sc_hd`, `sc_ls`, `sc_ms`)	7-track and 9-track MCU-oriented libraries	Generic 45 nm standard-cell library for academic / EDA prototyping
I/O Voltage Support	3.3 V typical; limited higher-voltage options	3.3 V and 6 V tolerant I/O options	1.2 V typical core; standard modern-node I/O
Strengths / Pros	Active open-source ecosystem; rich digital and analog cells; documented tapeout path; strong community support; mixed-signal friendly	Strong analog / I/O support; robust MCU-oriented process; predictable and mature device models; well-suited for high-voltage interfaces	Smallest feature size among the three; high density; faster switching speeds; better energy efficiency; representative of modern processes
Weaknesses / Cons	Larger geometry than advanced nodes; slower switching speeds; higher power for equivalent functionality compared to 45 nm	Larger geometry; slower switching speeds; less area-efficient; more limited open-source tooling and flows than SKY130	Generic PDK rather than a full production process; limited open-source ecosystem; often assumes commercial Cadence EDA tools for advanced flows
Best Use Case	Academic research, teaching, prototyping, mixed-signal ASICs, and open-source shuttle tapeouts	MCU design, embedded systems, high-voltage-tolerant blocks, and robust analog integration	High-performance digital ASIC prototyping, methodology and tool evaluation, and technology comparison studies
Community / Ecosystem	Large, active open-source community; strong OpenLANE and open-flow support	Smaller community; some open documentation but less community tooling than SKY130	Minimal open-source community; primarily used in Cadence training and academic EDA flows

3.7.3.4 PDK Selection

Cadence's 45 nm technology using the GSCLIB045 library is often the strongest choice for this project among these options because it provides the most modern process characteristics, the highest density, and the best overall performance-to-area ratio. As a 45 nm class technology, it offers significantly smaller transistors than either SKY130 or GF180MCU, which directly translates into faster switching speeds, reduced interconnect delay, lower dynamic power, and dramatically improved logic density. These advantages allow designers to implement more complex systems on a smaller silicon footprint while meeting aggressive timing and power targets that would be unattainable at 130 nm or 180 nm. Most importantly, the GSCLIB045 library is also highly consistent, well characterized, and tightly integrated with Cadence's commercial-grade EDA toolchain, enabling accurate timing closure, reliable synthesis, and predictable place and route behavior. This is especially important because the project focuses on the RTL to GDSII flow using the Cadence tool suite, and a reliable PDK reduces the time spent on debugging and allows the team to concentrate on implementation. Although Cadence's 45 nm PDK is generic and not tied to any real semiconductor foundry or manufacturable process, this is fully acceptable since the project is not intended for tapeout.

Given the complexity of the Cadence tools, it's best to use a tool that is well integrated and will cause the least problems.

3.7.4 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 [49].

3.7.5 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

3.7.6 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 [50].

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

3.7.7 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" [51]. 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.

3.7.8 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:

3.7.8.1 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.

3.7.8.2 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.

3.7.8.3 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.

3.7.8.4 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.

3.7.9 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 [52].

3.7.10 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, which can be shared across multiple cores, 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 [53].

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 [54].

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.

3.8 FuseSoC

3.8.1 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 [55]. 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.

3.8.2 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 and 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.

3.8.3 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)[56]. 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.

3.9 LiteX

LiteX is a Python-based framework for the rapid assembling of FPGA SoC fabrics [57]. Rather than acting primarily as a package manager, like FuseSoC [55], 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 strengths 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.

3.10 Bender (ETH Zürich)

Bender is a build and dependency management tool that allows hardware developers to organize and coordinate large sets of IPs in their designs[58]. In modern SoC projects, it is not uncommon for there to be dozens of modules, repositories, and components, each with its own required versioning. Without implementing a proper system to manage these dependencies, projects can quickly get disorganized, and Bender is able to fill this gap by acting as a simple, declarative, and version-controlled manager for hardware projects. Users must describe their project through a YAML manifest that specifies local sources, external dependencies, and version constraints. By using this YAML file, Bender can resolve these repositories and output file lists that work with most EDA tools.

Once the YAML file is parsed, Bender is able to create a representation of the project hierarchy while checking the validity of constraints. This process is very important as it guarantees that multiple users can work with a consistent and reproducible set of RTL sources. Finally, Bender resolves dependencies by traversing

through packages to ensure all required repositories are fetched and checked out before any build steps execute.

After dependencies are resolved, Bender needs to analyze the RTL sources to determine the necessary compile order. To do so, it scans for things like SystemVerilog packages and module references. Bender also uses its dependency graph to create a sorted list of RTL files and is usually stored as a .f file. The output can easily be integrated into workflows that use different tools (Verilator, Xcelium, Vivado, etc.), further adding to its versatility.

Bender also has the added benefit of supporting configuration parameters within the manifest mentioned above. These parameters allow projects to define optional IP blocks, feature flags, or variant-specific source inclusions. During the build process, Bender can go through the aforementioned parameters and decide on whether to include specific RTL files. This added feature allows for multiple related design variants to exist without needing to edit build scripts or maintain parallel filelists.

3.11 Chipyard

The Chipyard framework allows designers to build SoCs by using high-level descriptions instead of having to manually wire components [59]. It utilizes Scala configurations to specify the necessary CPU core, cache parameters, memory layout, and included accelerators and peripherals. Then, Chipyard uses the Rocket Chip infrastructure to convert these parameters into a full SoC structure, automatically handling interconnects, memory maps, and system-wide settings before any hardware is generated.

Once the system's configurations are specified, Chipyard uses the Chisel hardware description language to convert the high-level description into a low-level hardware representation. Each component in the system acts like a reusable generator that produces hardware whenever executed. With this intermediate design, it can be turned into Verilog modules and any desired simulation models. This layout does a great job of promoting modularity as developers can regenerate different SoCs without needing to edit RTL files directly.

Chipyard's build environment is primarily driven by sbt, which automates elaboration, Verilog generation, and the setup of simulation tools like Spike and Verilator. Along with this, it also integrates support for software tools to let users do things like compile bare-metal programs and run them on the generated SoC. These features make Chipyard a very powerful tool for rapid prototyping, but may prove to be too complex for our purposes.

Table 3.10: *Build/SoC Framework Comparison*

Tool	Role / Category	Implement In	Backends / Outputs	Strengths	Drawbacks
FuseSoC	IP package manager and unified build / orchestration for HDL projects	Python (with `.core` YAML metadata)	Wraps simulators and synth/P&R tools via Edalize (e.g., Verilator, Icarus, Questa/Xcelium; Vivado, Quartus, etc.); generates project files and runs flows	Non-intrusive; versioned IP cores; dependency resolution; tool-neutral abstraction; repeatable builds; aligns with Ibex documentation and flows	Not an SoC generator; learning curve for `.core` metadata; relies on backend tool availability and configuration; less opinionated, so project structure must be defined by the user
LiteX	Python SoC generator (stitches CPU, interconnects, DRAM/PCIe/Et peripherals; fast FPGA bring-up)	Python (Migen / Amaranth ecosys-tem)	Generates gateware and board projects for many FPGAs; integrates LiteDRAM, LitePCIe, LiteEth, etc.; produces bitstreams via vendor and open flows	Rapid FPGA prototyping; rich reusable "LiteX cores"; large board coverage; cohesive SoC composition	Skews toward FPGA / bitstream-centric flows; less tool-neutral for ASIC; opinionated stack; less suited to IP packaging as a first-class artifact
Bender	Dependency manager and reproducible RTL build specification	TOML manifests with a simple Rust-based backend	Resolves HDL dependencies, locks versions, and produces ordered file lists for simulators / synthesis tools	Minimal, fast, and reproducible; non-intrusive; easy to integrate into existing Make/CMake flows	Not a full build system; no native simulation or synthesis orchestration; focuses only on de-pendency/filelist management

Continued on next page

Tool	Role / Category	Implemen In	Backends / Outputs	Strengths	Drawbacks
Chipyard	RISC-V SoC generator and full research SoC framework	Scala / Chisel (with FIRRTL and Diplomacy)	Generates complete SoCs, simulation harnesses, Verilog, and FPGA/FireSim configurations	High-level hardware generation; modular SoC composition; rich ecosystem of RISC-V cores and peripherals; widely used in research	Steep Chisel/Scala learning curve; heavy dependencies; complex debugging; often overkill for simple designs or lightweight core integration

Chapter 4 - Standards and Design Constraints

4.1 Industry Standards

4.1.1 Cryptographic Standards

Modern secure processor design depends on a broad set of cryptographic standards that define how algorithms must behave, how keys are managed, and how hardware modules interact with the wider security ecosystem. These standards provide the rules that ensure cryptographic operations are mathematically sound, compatible with established protocols, and resistant to attacks that target both software and physical implementations. For a RISC-V security extension built on the Ibex core, understanding these standards is essential because the design must support secure boot, authenticated execution, protected key storage, and robust defenses against side-channel leakage. The following subsections highlight the major categories of standards that guided this project, covering general cryptographic module requirements, public key frameworks, and masking-based protections for secure hardware implementations. For this project, it is important to understand what industry standards are relevant to any ASIC design process as the extension is designed.

FIPS 140-2 is a United States federal standard that defines the security requirements for cryptographic modules used to protect sensitive information. It establishes four increasing levels of security, ranging from basic software or firmware protection to advanced hardware tamper resistance. The standard outlines requirements for cryptographic algorithms, key generation, key storage, physical security mechanisms, self tests, trusted roles, and secure module interfaces. It ensures that any cryptographic hardware or firmware behaves predictably, resists unauthorized access, and detects failures before performing secure operations. Although FIPS 140-2 has been succeeded by FIPS 140-3, it remains widely referenced in industry and continues to guide the design of secure embedded processors, hardware accelerators, and ASIC based cryptographic modules [60].

FIPS 197 is the NIST standard that specifies the Advanced Encryption Standard, a 128-bit block cipher supporting 128-, 192-, and 256-bit keys. It defines the Rijndael-based round structure, key schedule, and test vectors that ensure interoperable implementations across hardware and software platforms. Because AES is mandated or strongly recommended in many higher-level profiles and protocols, it serves as the default symmetric primitive for secure boot, encrypted storage, and authenticated execution in modern processors. In this project, AES is the assumed block cipher for any symmetric cryptographic acceleration or DRBG constructions (e.g., CTR-based generators), allowing the Ibex security extension to align with existing cryptographic libraries and FIPS-oriented validation flows [61].

IEEE P1363 is a broad public key cryptography standard that defines general mathematical frameworks, operation formats, and implementation guidelines for a wide range of public key techniques. It covers algorithms based on integer factorization, discrete logarithms over finite fields, elliptic curve systems, and related key agreement and digital signature schemes. The standard provides unified notation, recommended parameter sets, and definitions for operations such as key generation, signature creation, signature verification, and secure key exchange. Its purpose is to ensure consistent, interoperable, and secure public key implementations across hardware and software platforms. For a RISC-V security extension, IEEE P1363 serves as an important general reference because it outlines how public key operations should behave regardless of the specific algorithm or curve used, making it a foundational guideline for designing cryptographic accelerators or instruction set extensions [62].

NIST IR 8214A provides formal guidance on threshold implementations of cryptographic algorithms, which is the theoretical foundation for Boolean masking in hardware. The document introduces the core principles of threshold schemes, including non-completeness, correctness, and uniformity, which together ensure that intermediate computations remain statistically independent of secret values even in the presence of glitches. It describes how multi share Boolean masking should be constructed, how randomness must be incorporated, and how hardware must avoid secret dependent transitions that leak information through side-channels. Because this project uses Boolean masking to harden the Ibex security extension, it was essential to understand the industry standards that define how masking must be implemented and evaluated. NIST IR 8214A served as a reference for ensuring that masked operations follow accepted cryptographic engineering practices and align with modern expectations for side-channel resistance in secure ASIC designs [63].

NIST SP 800-90A defines the deterministic side of random bit generation, specifying a family of Deterministic Random Bit Generators (DRBGs) that expand a high-entropy seed into long streams of pseudorandom bits suitable for cryptographic use. The standard describes three approved DRBG mechanisms, Hash_DRBG, HMAC_DRBG, and CTR_DRBG, each built from well-studied primitives such as hash functions, HMAC constructions, or block ciphers (e.g., AES). For each mechanism, SP 800-90A prescribes how to instantiate the generator, how often it must be reseeded, how to request prediction resistance, and how much security strength can be claimed based on the underlying primitive and seed length. In the context of a secure RISC-V processor, these DRBGs provide the reference model for how on-chip pseudorandom streams should be generated for keys, nonces, and masking values, ensuring that any randomness-dependent defenses are grounded in a well-defined, widely reviewed standard [29].

NIST SP 800-90B focuses on the entropy sources that feed DRBGs, providing design principles, statistical tests, and documentation requirements for physical or system-level noise sources used in Random Bit Generators. Rather than specifying a particular circuit or implementation, the standard describes how to model entropy sources as either IID or non-IID processes, how to characterize their min-entropy using conservative estimators, and how to deploy health tests to detect catastrophic failures or severe bias over time. It distinguishes between the raw noise source and optional conditioning components (such as cryptographic hash functions) that can improve statistical properties before seeding a DRBG. For a secure Ibex-based ASIC, SP 800-90B is the primary reference for how any on-chip TRNG or noise-based entropy source must be evaluated and monitored before its outputs are trusted to seed the masking and key-generation mechanisms used by the extension [28].

NIST SP 800-22 provides a statistical test suite for evaluating whether the outputs of random number generators and pseudorandom generators exhibit properties consistent with randomness when viewed through classical hypothesis testing. The document defines a collection of complementary tests (including frequency, runs, block frequency, approximate entropy, linear complexity, and random excursions) that each target different structural patterns that might reveal bias or predictability in a bitstream. Implementers apply these tests to large collections of sample sequences and interpret the resulting p-value distributions against a chosen significance threshold, using failures as evidence that a generator may be misconfigured or fundamentally flawed. Within this project, SP 800-22 offers a standardized way to report the "apparent randomness" of prototype entropy sources and DRBG outputs, complementing the more structural guarantees from SP 800-90A and SP 800-90B and helping to validate that mask values and keys do not exhibit obvious statistical weaknesses [31].

Table 4.1: *Summary of Relevant Cryptographic Standards*

Standard	Focus Area	Relevance
FIPS 140-2	Requirements for validated cryptographic modules, algorithms, and key handling.	Ensures secure module behavior, consistent operation, and trusted cryptographic processes.
FIPS 197 (AES)	Standardized 128-bit block cipher with 128-, 192-, and 256-bit keys.	Baseline symmetric primitive for secure boot, encrypted storage, and masked or accelerated cryptographic operations in the Ibex-based extension.
IEEE P1363	General framework for public-key systems including RSA, ECC, and discrete-log schemes.	Provides consistent rules for key generation and signatures, supporting interoperable instruction design.
NIST IR 8214A	Guidelines for threshold implementations and Boolean masking for side-channel resistance.	Defines masking rules, non-completeness, and glitch resistance needed for secure masked Ibex hardware.
NIST SP 800-90A	Deterministic random bit generators (DRBGs) built from hash, HMAC, or block-cipher primitives.	Specifies how pseudorandom keys, nonces, and masking values should be generated from high-entropy seeds in compliant secure processors.
NIST SP 800-90B	Entropy sources used for random bit generation, including design, testing, and health monitoring requirements.	Guides the design and validation of on-chip noise sources that seed DRBGs and underlie the randomness used by the Ibex security extension.
NIST SP 800-22	Statistical test suite for evaluating the apparent randomness of RNG and PRNG outputs.	Provides standardized metrics for assessing prototype RNG and DRBG outputs, helping confirm that masking streams and keys lack obvious statistical weaknesses.

4.2 Design Standards

4.2.1 SystemVerilog and RTL Coding Standards

The IEEE 1800 SystemVerilog standard defines the language rules, coding practices, simulation semantics, and synthesizable constructs used to build modern digital hardware. It provides the foundation for writing reliable and consistent register transfer level (RTL) logic, including conventions for module structure, clocking, reset behavior, and signal naming. Following SystemVerilog best practices helps ensure that the security extension integrates

cleanly with the existing Ibex core and follows predictable timing and synthesis behavior. The standard also defines constructs such as assertions, always blocks, enumerated types, and interfaces, all of which support clearer design intent and reduce the likelihood of functional errors in a masked cryptographic pipeline [64]. Adhering to these RTL coding standards was important for producing hardware that is modular, synthesizable, and compatible with industry-grade verification tools.

4.2.2 Low Power and Low Switching Activity Standards

Low power design guidelines provide practical rules for reducing dynamic and static power consumption in ASICs while maintaining correct functional behavior. These standards include clock gating, minimizing unnecessary switching activity, reducing glitch propagation, and using balanced logic structures to avoid power spikes that could reveal sensitive information. In security focused hardware, low power design is tightly linked to side-channel mitigation because switching activity correlates with power analysis leakage. For the Ibex security extension, applying low power design principles helped maintain efficiency on a resource constrained embedded core while also supporting the masking strategy by reducing data dependent transitions. These standards shaped the architecture of the masked logic, register updates, and random refresh circuitry to maintain both energy efficiency and resistance to power analysis attacks.

4.2.3 ASIC Synchronous Design Standards

In ASIC design, there are specific, strict, and well-defined rules that engineers in the industry must follow to guarantee timing correctness, safe operation, and reliable behavior. To ensure the project meets typical industry standards, the following standards were researched in order to inform the team of best practices when designing.

- Register-to-register Timing Closure Standards: a methodology involving static timing analysis (STA) to ensure that data arrives at its destination register within the required time.
- Glitch-Free Combinational Logic: establishes rules for minimizing glitches, such as using balance logic, inserting pipeline registers, and avoiding long XOR or adder chains. This is critical for boolean masking because glitches can introduce measurable leakage.
- Synchronous Reset Architecture Standards: defines how reset signals must behave, including synchronous reset deassertion, clean initialization, and predictable register states [65].
- Randomness Integrity: establishes design requirements for securely integrating randomness sources in hardware, including synchronizing random bit updates, preventing feedback loops, and avoiding combinational dependencies.
- Switching Activity Minimization: defines rules for reducing dynamic power and data-dependent switching, such as using balanced routing and avoiding asymmetric logic on masked shares.

Table 4.2: *Comparison of Relevant Design Standards*

Standard	Focus Area	Relevance
SystemVerilog (IEEE 1800)	Defines RTL coding rules, module structure, clocking, and reset behavior.	Ensures clean, synthesizable RTL that integrates safely with the Ibex core.
Low Power / Switching Activity	Guidelines for reducing dynamic power, balancing logic, and limiting data-dependent transitions.	Supports both efficiency and reduced leakage in masked and noisy datapaths.
ASIC Synchronous Design Rules	Timing closure, glitch-free logic, proper reset design, and stable clocking.	Guarantees reliable behavior of masked logic, registers, and noise generation circuitry.

4.3 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.

4.3.1 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)*

A defining characteristic of the base RISC-V design is its 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 [66].

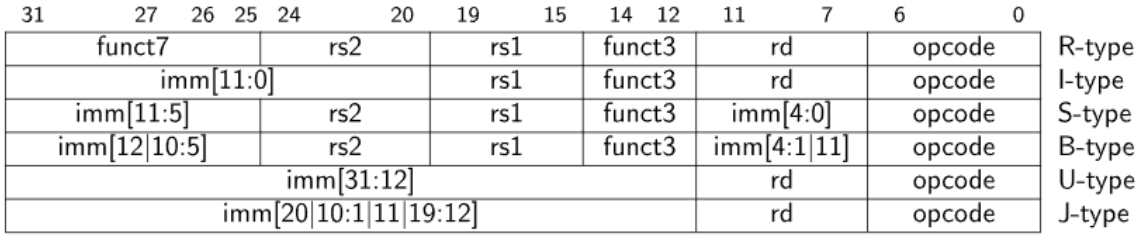


Figure 4.2: *Instruction Types for RV32I ISA*

4.3.2 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.

4.3.2.1 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.

4.3.2.2 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 (AMOs) 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.

4.3.2.3 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)*

4.3.2.4 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 [67].

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.

4.3.2.5 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.

4.3.3 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 [68].

4.3.4 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.

4.3.5 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 [69].

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.

4.4 Verification and Validation Standards

4.4.1 Functional and Code Coverage

Functional coverage is a verification metric used to check whether all intended behaviors, scenarios, and design features have been exercised during simulation. Instead of focusing on the structure of the RTL itself, functional coverage is tied to design intent and specification requirements. It essentially ensures that you're testing everything you need to, to ensure your design works as expected. Functional coverage covers the following:

- Every custom instruction was executed under all valid conditions.
- Masked datapaths exercised all combinations of mask shares.
- Random refresh behavior was triggered.
- Ensures that an LFSR was used across its entire period or range.
- Pipeline stalls, flushed, and control hazards interact correctly with masked operations.
- Corner cases (boundary values, illegal opcodes, invalid CSR writes) were tested.

Functional coverage is implemented using SystemVerilog covergroups, coverpoints, and crosses that record events and interactions during simulation. This metric is especially important in secure hardware because masked implementations must be verified to ensure that all share interactions, randomness updates, and state transitions operate correctly and do not introduce security weaknesses.

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 [70].

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 [71]) 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 monitor, 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.

4.5.1 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 [70].

4.6 OpenTitan

OpenTitan is not only a prominent open-source silicon Root-of-Trust (RoT) implementation; it also codifies a reusable process for IP design and verification that other projects can adopt. In particular, the Comportable IP (CIP) specification defines common interfaces, register conventions, and integration rules for TileLink-UL (TL-UL) peripherals, while the companion CIP UVM library standardizes IP-level testbench architecture and reusable sequences. These materials are complemented by style guides (SystemVerilog/Verilog, C/C++, HJSON) and a set of hardware development/verification stages that establish project-wide maturity gates and documentation expectations. Together, these artifacts function as "living standards" that improve portability, auditability, and DV quality across independently developed IPs [72, 73].

4.6.1 Comportable IP (CIP) Architecture

The Comportable IP specification prescribes what every on-chip peripheral must "look like" to the rest of the system. Each IP exposes a TL-UL device interface with well-defined clocking and reset semantics and avoids ad-hoc bus protocols. Control and status registers (CSRs) are defined in HJSON and auto-generated into consistent RTL register files, software headers, and UVM RAL models, ensuring that field access types (e.g., rw, ro, wo), reset values, error behavior, and lock/enable (regwen) semantics are uniformly implemented and documented. Comportable also distinguishes functional interrupts from security-critical alerts: interrupts report normal operational events to software, whereas alerts are differential, tamper-relevant signals routed to the SoC's alert_handler for classification and potential escalation if software fails to respond. Beyond these surface interfaces, CIP encourages explicit hooks to system security services (e.g., lifecycle/OTP, entropy sources) and "design for diagnosability," so that both functional and security behaviors can be exercised deterministically in DV. This architectural discipline yields IPs that integrate predictably into the SoC and are testable with a common methodology [72, 27].

4.6.2 CIP Library Classes

To ensure verification consistency, OpenTitan provides a reusable UVM stack (the CIP library) that supplies the standard anatomy of an IP-level testbench. A typical environment instantiates cip_base_env, a virtual sequencer coordinating TL transactions and register operations, and a parameterizable scoreboard that mirrors the UVM RAL model to check end-to-end behavior. Common sequences (e.g., CSR "smoke" and read-write campaigns, reset/interrupt/alert exercises, TL access tests) are available to bring new IPs to a baseline quality bar with minimal bespoke code. Protocol assertions for TL-UL, register consistency checks, and coverage models plug in as first-class components, while code structure adheres to the project's DV style guides to keep agents, sequencers, and configuration objects uniform across IPs. Because RAL and headers are auto-generated from the same HJSON source used by RTL and software, the testbench and design remain synchronized, reducing drift and enabling reliable reuse [72, 70, 71].

4.6.3 Security Verification in CIP Library

Security-relevant behavior is verified with the same composable infrastructure rather than via one-off tests. At the IP level, tests intentionally stimulate conditions that must raise alerts, then observe the differential alert signaling and verify downstream handling by alert_handler at the SoC level. Expected outcomes include timely interrupt delivery, programmable classification, and, if unhandled, escalation actions as documented in the system's theory of operation. Protocol correctness (e.g., TL-UL response rules, no X-propagation on control) and CSR integrity (shadowed registers, regwen guards, error paths) are enforced through assertions and scoreboard checks, treating "security by correctness" as a measurable requirement. Evidence is organized against the project's verification stages (e.g., V1→V2→V3), which define coverage targets, checklist items, and documentation artifacts required to claim maturity. This process orientation ensures that security properties are captured as testable contracts with traceable coverage, not merely as aspirational design goals [73, 72].

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 0.9 V to 1.2 V provides a stable operational range that balances energy efficiency, timing reliability, and side-channel consistency. Voltages below 0.9 V increase the risk of timing violations and signal degradation, while those above 1.2 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 15 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 modified core achieves strong practical resistance to side-channel attacks while preserving timing determinism and engineering feasibility.

4.7.8 Chip Density

Chip density is important to the physical design process of an ASIC. The higher the chip density, the smaller the total die size. In theory, this would reduce manufacturing and material costs. However, ensuring your chip utilization is not too high is critical for maintaining the functionality of your design.

A chip density that is "too high" (75-80%) may cause routing issues, decreased power integrity, design rule violations, and degraded timing performance.

Choosing a density just above 60% leaves sufficient space for routing resources and helps minimize manufacturing costs without compromising chip efficiency.

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.

4.8 Practical and Security Constraints

4.8.1 Design Constraints

In this project, the design constraints describe the limits imposed by the target technology, the Ibex micro architecture, and the security features implemented. They capture what the design must not violate in terms of timing, area, power and integration behavior. These constraints are catered towards implementation of Boolean Masking, noise generation, and clock randomization. Understanding these constraints early in the project ensures that the extension remains synthesizable, can emulate deterministic behavior at the target clock frequency, and still behaves as intended. Some of these crucial constraints are expanded upon below.

Clock frequency and timing budget for the Ibex two stage pipeline- Even in simulation, Ibex assumes a maximum timing budget for ALU operations and masked logic. If the added masked datapath is too slow, the design may violate cycle timing, extend critical paths, or fail functional simulation due to incorrect pipeline sequencing.

Power and switching activity limits for randomness and masking logic- continuous random number refreshing, noise injection, and masked computation increase switching activity, which in a real ASIC would raise dynamic power. Excessive switching also risks additional simulated side-channel leakage if not carefully balanced.

Datapath width and register file constraints of the RV32I architecture- the RV32I ISA requires a fixed 32 bit register width, a narrow ALU, two read ports, one write port, and limited CSR availability. Adding masking or randomness features cannot alter these architectural constraints without breaking ISA compatibility or disrupting Ibex pipeline semantics [74].

Integration constraints between new instructions, CSRs, and Ibex pipeline stages- custom masking instructions, control registers, and noise control signals must integrate cleanly with Ibex's decode, execute, and writeback stages. Incorrect integration could introduce pipeline hazards, break existing CSR behavior, or violate RISC-V privileged specifications.

Randomness Number Generation Constraints - Given that the focus of this project is to leverage to RTL-to-GDSII flow, true random number generation is not feasible within the RTL-to-GDSII flow, as it would require analog elements or custom designs developed outside the RTL. Consequently, true randomness cannot be achieved by synthesizing RTL alone. True randomness cannot be achieved by writing and synthesizing RTL. The only way of accomplishing this would be implementing something outside of the RTL-to-GDSII flow.

Randomness Number Quality for Boolean Masking - In order for the original data to be statistically independent of the shares generated from Boolean Masking, the generation of high-quality random numbers is required. High-quality randomness is essential

because Boolean masking assumes that the mask values are unpredictable, uniformly distributed, and independent of each operation. If the random numbers are biased, insufficiently random, or inadvertently reused, statistical leakage can reemerge. Even small deviations from uniformity can allow an attacker to reduce the search space or exploit predictable patterns when correlating power or EM traces. In practical attacks, adversaries often rely on subtle statistical imbalances, and weak randomness can amplify these leakages to detectable levels.

Moreover, side-channel adversaries can collect millions of traces. With enough data, even a tiny correlation between the masked value and the physical leakage becomes exploitable. If the random number generator cannot keep up or does not provide statistically independent values, masking security degrades, especially under simulated side-channel conditions.

Secure Gate Constraints - The secure gates used to implement Boolean Masking in this processor consist of secure versions of a NOT, AND2, OR2, and XOR2 gate as well as a D flip flop. Each of these secure cells is constructed from the primitive gates available in the underlying standard cell library, and they form the only set of logic elements permitted in the processor design. This creates a very narrow selection of usable circuitry compared with the full GSCLIB045 library that will be employed during synthesis, and this restriction becomes a critical design constraint. Since the processor can only be built from this small secure subset, the synthesis tool must synthesize the design with only the non-secure counterparts of the secure gates before replacing them with the secure version, which naturally affects how effectively it can generate an efficient gate level implementation.

The first major impact concerns the flexibility of logic mapping. Standard cell libraries are designed with a wide spectrum of gates, including simple NAND and NOR cells as well as more complex structures such as multiplexers, AOI and OAI gates, and specialized arithmetic cells. These complex cells often provide compact implementations of multi level logic. If they are not available, the synthesis tool must recreate the same functions using a larger number of basic gates. This increases the total number of gates, expands the area footprint, and typically increases both static and dynamic power consumption.

Timing is also affected. Complex cells are often created to shorten critical paths by collapsing multiple logical operations into a single stage. Without them, the resulting circuit may require more logic depth, which increases propagation delay. This makes it harder to reach the desired clock speed. The synthesis tool may attempt to compensate by using larger drive strengths from the permitted subset, but this often increases power and introduces additional placement pressure for physical design.

The quality of optimization is influenced as well. Many synthesis optimizations rely on having a diverse set of cells that enable efficient restructuring of logic. For example, algebraic simplifications may depend on the availability of three input or four input gates, certain multiplexer variants, or other compound structures. Restricting the cell library reduces the number of transformation choices, which limits how well the tool can balance performance, area, and power. Some optimizations may become entirely infeasible.

Signal integrity and design robustness can also suffer. Standard cell libraries usually include cells tailored for specific tasks such as handling high fan out, reducing glitch propagation, or providing low noise transitions. If these cells are unavailable, the resulting design may experience greater timing variability or increased glitch activity. During place and route, back end tools may struggle to meet buffering or loading requirements if buffer choices or threshold voltage variants are missing.

Overall, limiting the synthesis process to a subset of gate types restricts architectural choices, complicates timing closure, reduces optimization freedom, and constrains back end design strategies. As a result, the final synthesized module may have worse power, performance, and area metrics and may deviate significantly from the intended design objectives.

4.8.1.1 Software Constraints

The software tools used throughout this project were influenced by the resources available to us through our sponsor. Since the project is intended to follow an industry aligned ASIC design flow, the team relied primarily on Cadence tools, including Xcelium for simulation, Genus for synthesis, and Innovus for backend preparation. These tools define many of the practical constraints of the design process, such as supported HDL constructs, simulation performance limits, synthesis optimizations, and the accuracy of timing or power estimates. Although Cadence served as the primary environment, the team also incorporated open-source resources such as EDA playground and RISC-V software toolchains when appropriate. This mixed environment required ensuring consistent behavior between commercial and open-source tools, and it shaped how the RTL, testbenches, and verification infrastructure were implemented.

4.8.1.2 Fabrication Constraints

While the project followed an ASIC style workflow, full chip fabrication is not within scope due to cost, time, and the intended outcomes per the sponsor. Instead, the goal was to advance the design through RTL development, synthesis, verification, and layout level exploration without committing to expensive masks or foundry production. Fabrication costs for modern technology nodes are far beyond the needs of this academic and research focused effort, and manufacturing a secure core would require extensive physical verification, silicon characterization, and security testing that exceed project constraints. Therefore, the design was carried only to the layout and physical planning stages, allowing the team to evaluate area, timing, routing complexity, and integration feasibility without incurring the requirements of a full tape out.

4.8.2 Practical Constraints

4.8.2.1 Time Constraints

This project is designed to be completed over the course of a two-semester Senior Design sequence. Given this timeline, the workload of our team, and the time required to become proficient with advanced ASIC design tools, we established clear target goals and stretch goals that reflect the practical limits of the schedule.

A major contributor to the project timeline is the requirement to use the Cadence Design Suite, including tools such as Xcelium, Genus, and Innovus. These tools are industry standard and highly sophisticated, which means they require substantial training before they can be used effectively. As a result, a significant portion of Senior Design 1 was dedicated to completing the required Cadence training modules. Developing competence in these tools is also a primary objective of the project and aligns with the expectations of our sponsor, so prioritizing time for training was essential.

Team scheduling also played a role in our timeline. Several group members are enrolled in demanding coursework, including graduate-level classes, which reduced the number of time windows where all members could meet. With a group of five, finding common availability during the week was challenging, so we added a secondary weekly meeting on Saturdays to ensure consistent progress.

Finally, ASIC design projects are inherently complex and, in industry, can require years to complete before reaching full tapeout. With this in mind, our project goals were intentionally scoped to emulate key stages of the ASIC design flow rather than attempting a full, end-to-end development cycle. Through coordination with our sponsor and review of previous Senior Design projects, we identified milestones that were both meaningful and achievable within two semesters. This ensured that the project remained rigorous while still attainable.

4.8.2.2 Economic Constraints

The economic constraints of this project are shaped primarily by the high cost traditionally associated with ASIC design and fabrication. In industry, developing a custom integrated circuit can require significant financial investment, including licensing fees for professional design tools, compute resources for simulation, and expenses associated with full chip fabrication. However, because this project is sponsored, our team is provided access to the Cadence Design Suite at no cost. This eliminates what would normally be one of the largest economic barriers for academic or small-scale ASIC projects, since tools like Xcelium, Genus, and Innovus can collectively cost hundreds of thousands of dollars in commercial settings. The support from our sponsor allows us to focus on design, verification, and learning the toolflow without the financial burden typically associated with acquiring professional-grade EDA software.

Beyond software licenses, the project remains economically constrained by the fact that full ASIC manufacturing is prohibitively expensive and far beyond the budget of a university design team. Modern fabrication processes, even at older technology nodes, involve high mask costs and long fabrication cycles that are not feasible within the scope or resources of a student project. As a result, the project is limited to RTL design, simulation, synthesis, and layout-level exploration without advancing to tape-out. This limitation is consistent with the educational goals of the capstone experience, which prioritize understanding the ASIC design flow rather than producing a fabricated chip. By staying within simulation and physical design stages, the project remains economically sustainable while still offering meaningful exposure to industry-relevant methodologies.

4.8.3 Security Specific Constraints

Security was the primary focus of this project, which introduced a unique set of constraints that directly shaped the design, scope, and implementation of the hardware extension. Since the goal of the project was to improve resistance to side-channel and fault injection attacks, the extension required features such as Boolean masking, random refresh logic, and controlled noise generation. Each of these additions introduced certain limitations that had to be understood and respected throughout the design process. These constraints ensured that the security mechanisms operated correctly, did not interfere with Ibex pipeline behavior, and did not introduce new leakage pathways.

One major constraint was the requirement for high-quality randomness, since masking schemes rely heavily on fresh, unpredictable random values. The design had to ensure that the LFSR and noise generation units provided randomness at the rate required by the masked datapath without causing stalls or reuse of mask shares. In addition, glitch resistance placed restrictions on combinational logic depth, forcing careful structuring of masked operations to avoid signal propagation differences that could leak sensitive information. Similarly, the need for non-interference with unmasked logic required that masked datapaths, refresh controllers, and noise sources could be added without altering the timing behavior or control flow of the Ibex core. Avoiding secret-dependent timing, preventing data-dependent switching, and ensuring constant-time execution all constrained how instructions, registers, and datapath elements were implemented within the RV32I architecture.

These security constraints influenced many of the architectural choices in the project. To satisfy the randomness requirements of Boolean masking, noise insertion, and internal state refreshing, the design incorporates multiple Pseudo Random Number Generator (PRNG) mechanisms, including a Linear Feedback Shift Register (LFSR), a Cellular Automata Shift Register (CASR), and a Gaussian Noise Generator Verilog IP core sourced from OpenCores. Each generator serves a different role within the architecture, allowing the system to maintain independent and diverse randomness paths while staying fully synthesizable and compliant with RTL design constraints. This approach provides sufficient entropy for simulation-level validation and mask refreshing, while avoiding the complexity and fabrication dependencies of integrating a true hardware random number generator.

4.8.4 Verification and Validation Constraints

Verification for an ASIC design introduces several technical constraints that limit how thoroughly the system can be tested within the available project timeframe. A key constraint is timing accuracy in simulation. While RTL simulation does not fully reflect post-layout timing behavior, the verification environment must still approximate the intended clock frequency and datapath timing of the Ibex core. If the assumed timing diverges from the synthesized design, functional tests may pass incorrectly, or timing-related corner cases may remain undetected. Accounting for these timing assumptions in our testbenches is necessary to ensure that the security extension behaves correctly under realistic operating conditions [75].

Another important constraint stems from tool capacity and performance limitations. Adding masked datapaths, noise generators, and additional control logic increases overall design size and switching activity, which slows down simulation tools such as Cadence Xcelium. Long waveform capture, repeated regression runs, and coverage-driven testing can further compound simulation time. Because verification requires running extensive tests to reach meaningful coverage, we must structure the test environment efficiently by modularizing components, limiting waveform dumping, and avoiding unnecessary full-system simulations that consume time or risk tool instability.

Finally, verification is constrained by the manual effort required to develop and maintain testbenches. Writing assertions, constructing corner-case scenarios, debugging failing tests, and validating functional behavior across many masked states all require significant engineering time. In industry, full verification cycles can span months or years; however, this project must be completed within two semesters. As a result, our verification scope must be carefully targeted, prioritizing correctness of masking behavior, functional

accuracy of new instructions, and integration with the Ibex pipeline, while acknowledging that exhaustive industry-level validation is beyond the limits of the project timeline.

4.9 Industry Constraints

4.9.1 Economic Constraints

Economic constraints define the financial limitations and cost-related considerations that influence the scope, tools, and deliverables of an ASIC design project. Although this project does not require any financial support from the student team, real-world ASIC development involves significant expenses for licensing, compute resources, fabrication, verification, and professional labor. Understanding these constraints is important for setting practical expectations and designing a project that is realistic within an academic environment. Additionally, it is important knowledge to obtain as we step through the ASIC design process, and prepare ourselves for work in the industry.

One of the largest economic factors in ASIC design is the cost of industry-level Electronic Design Automation (EDA) tools. In industry, access to tools such as Cadence Xcelium, Genus, Innovus, and Virtuoso requires expensive, annually renewed licenses. For this project, these tools were provided by our sponsor at no cost, significantly reducing the financial burden on the team. Without sponsorship, the cost of using these tools alone would make a project of this scale impractical in an undergraduate setting.

Another significant economic constraint is the cost of fabrication. Producing a physical ASIC, even at older technology nodes, can cost tens of thousands of dollars for prototyping and significantly more for volume production. Fabrication is far beyond the budget and scope of a senior design course. Therefore, this project is limited to the RTL-to-GDSII flow, stopping at layout generation rather than proceeding to tapeout. This constraint shapes the project by focusing our efforts on synthesis, functional verification, physical design, and layout generation rather than full chip manufacturing.

Finally, economic constraints include the human effort, time, and opportunity cost associated with advanced design work. Verification and physical design are notoriously labor-intensive stages, and in professional settings require large engineering teams. Within a student project, this complexity must be reduced to manageable chunks to avoid excessive workload. For this reason, the project's goals were chosen to balance educational value with realistic time and effort limitations.

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, 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 softwares 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 - LowRISC Toolchain Setup (DV & Co-simulation)

While it has been actively discussed that this project will see the usage of specific Cadence tools, paired with open-source toolchains and projects, it is also vital to recognize the environment in which these tools are invoked. Many of these tools are designed and intended to be used within a Linux environment. This is more of an implicit requirement of this project, and not one that has been laid out with many resources for tackling challenges regarding toolchain bring-up or implementation. Alongside this, many members of our team do not have very significant experience in Linux/Unix environments, meaning that executing necessary commands to get features functioning in the workflow may be difficult. For these reasons, LLMs (specifically ChatGPT 5.1) have been invaluable tools for setting up this "infrastructure".

Specifically, we recognized that the Ibex core is packaged with a configurable Design Verification (DV) and Co-simulation environment which leverages other dependencies (such

as Spike, a fork of a RISC-V ISA simulator). The setup of these dependencies is well-documented within the Ibex GitHub repository; however, it assumes that users will have administrator access to install all dependencies and system-level packages. Along with this, it is typical for VLSI tools to be used on Linux distributions such as Ubuntu 22.04/24.04. These are luxuries that we are not presented with on the UCF VLSI servers. Instead, we are presented with a Red Hat/Fedora-based distribution running relatively old system packages.

ChatGPT was initially prompted with the Ibex DV/Co-sim repository page contents, in which it specified the main steps required to build and invoke this environment. It was quick to provide insight on how to run commands such as "sudo apt update" and "sudo apt upgrade", as a simple means to get core system packages up and running. Running "sudo" commands without any kind of administrator consent would be extremely irresponsible and dangerous to do, so it was deemed necessary to prompt ChatGPT about this limitation. After more conversation with ChatGPT, it was realized that the VLSI server contains all of the packages necessary to run the toolchain, albeit at older versions.

Perhaps the most notable was Python 3: the UCF VLSI server provided version 3.6.8; however, the Ibex tool flow needed at least version 3.10. ChatGPT was able to point out that it is possible to make local installations of different Python versions and create a virtual environment (venv) of this specific Python build, allowing us to both use the necessary Python version for Ibex DV/Co-sim and also avoid mixing our (potentially) newer Python libraries with any already existing within the VLSI server's directories. Beyond this, many other dependencies were installed using Python from the provided "requirements.txt" file within Ibex, or simply through commands that were local to our user profile and didn't need any kind of administrator access. One of the key things that the Ibex documentation pointed out was that system path variables (e.g., $PYTHONPATH) need to be assigned in order for the toolchain to work correctly. Many of these were set during the installation process, but this only assigns them for that session. ChatGPT provided export commands to include either within a bashrc file or a separate script that would assign these variables with ease, simplifying the initialization process of attempting to use the toolchain. At this point, ChatGPT was sufficient to walk us through the Linux environment setup of the Ibex DV/Co-sim toolchain. This conversation with ChatGPT may be continued in order to develop an all-encompassing "compilation script" that would automate the entire build process of this toolchain (from Python venv setup all the way to bash script setup of environment variables) to allow readers to set up the exact same toolchain flow and file structure used within this project.

5.3 Example 3 - Generating RTL hierarchy order for Synthesis Scripts

To run synthesis, it is important to update and maintain the project's Tcl scripts because these scripts control the sequence of steps within the RTL to GDS flow. A central responsibility of the synthesis Tcl script is to ensure that all RTL files are read in a correct and deterministic order. The elaboration stage of the synthesis flow depends heavily on this ordering because the tool must resolve the design hierarchy from the bottom up. If files are processed in the wrong order, such as reading top level modules before their submodules, the synthesis tool may fail or generate incomplete hierarchies. For this reason, SystemVerilog files representing modules that sit lowest in the hierarchy, meaning modules that instantiate

no other modules, must be read first. Higher level blocks and finally system level top modules follow.

For the small configuration Ibex core, this process was relatively straightforward because the Ibex documentation included a reference file list that already described the recommended ordering of its RTL files. This allowed the project's Tcl scripts to be updated quickly and with confidence. The situation was quite different for the Ibex Simple System SoC. The Simple System does not provide a pre determined or officially maintained file ordering, and it integrates a more diverse set of modules including RAM blocks, a timer module, a bus interface, and a traced version of the Ibex core. Identifying the correct bottom up synthesis order manually would have required examining each file in detail and determining every direct and indirect dependency. Although this manual approach was possible, it would have been time consuming for the design team.

To accelerate the process, the project incorporated the use of ChatGPT through GitHub Copilot in an attempt to automate the discovery of the RTL hierarchy. ChatGPT was supplied with the full set of Simple System RTL files and prompted to infer the correct bottom up ordering. This type of task aligns well with the strengths of a language model because it involves pattern recognition, identification of module instantiations, and the generation of structured output. ChatGPT successfully inspected the provided filenames, identified module names, and constructed a plausible file ordering. It also produced a text file containing the proposed hierarchy, formatted so that it could be inserted directly into the synthesis Tcl script.

Although the initial output from ChatGPT appeared reasonable, the true measure of success was whether the design would synthesize correctly. After the new file list was incorporated into the Tcl script and the synthesis flow was executed, the tool failed early in the elaboration phase. This failure was traced to the omission of several primitives and supporting files. Some of these missing elements originated from vendor libraries or internal primitive directories that were not explicitly referenced within the RTL files supplied to ChatGPT. Because ChatGPT only received the subset of files chosen for the prompt and did not have full visibility into the entire repository, it could not identify all of the external dependencies required for successful synthesis.

Additional complications arose from the use of the Ibex small configuration. This configuration disables features such as the instruction cache, secure Ibex support, and several ISA extensions. ChatGPT attempted to incorporate these details and was partially successful in excluding many non synthesizable files associated with disabled features. However, parameter driven hardware designs often involve subtle interactions where one parameter enables or disables functionality in files that are not obviously linked. As a result, ChatGPT's inferred file hierarchy occasionally excluded modules activated indirectly by configuration parameters.

From this experience, the project gained insight into how large language models can support hardware development workflows. ChatGPT proved useful for accelerating organizational tasks such as identifying possible module hierarchies or drafting a candidate file list. However, the tool could not replace verification by synthesis tools or manual inspection of RTL sources. A language model can generate structured and plausible output, but without access to the full project context, it cannot guarantee correctness.

In the end, ChatGPT provided the project with a valuable starting point that reduced the initial effort required to assemble a file list for the Simple System. The generated output still required correction and refinement, but the process highlighted which components the

assistant considered leaf modules and which it treated as higher level constructs. More importantly, this exercise demonstrated the importance of iteration and human oversight when integrating AI generated results into hardware design pipelines.

5.4 Example 4 - EDA Tool Comparison for an ASIC Design

As part of the research and design process for this project, it was useful to explore more commonly used EDA tools and resources. Although we had known from the beginning that Cadence Design Suite would be our utilized software, since we were provided the licensing from our sponsor, it was important to delve into other softwares for a few reasons: researching other tools is crucial to making informed decisions in any project, it will benefit our depth of knowledge going into the rest of the project, and it allows us to emulate a more conclusive design process.

Initially, after prompting ChatGPT to provide me with other ASIC design tools for comparison, it gave me seven tools and explained each of their roles in detail. The first response helped me understand the broader EDA landscape and how different categories of tools fit together, such as simulators, formal verification engines, physical design flows, and security-focused verification frameworks. After refining the prompt, and asking specifically for the most common industry grade tools, ChatGPT provided a more practical list for what I was looking for. It also broke down what companies actually use day-to-day, why certain tools are more frequently used for certain steps in the design process, and how they compare to each other. This part was especially helpful, because it framed the project in a realistic engineering workflow, showing how professionals would approach the different design steps.

In the final prompt, I asked whether there was still useful information to learn about the other competing tools even though I would only be using Cadence. ChatGPT gave an in depth explanation of why this knowledge still matters. It highlighted how different ecosystems like Synopsys and Siemens influence industry expectations, how tools such as PrimeTime or Fusion Compiler shape timing and backend methodologies, and how specialized security tools like Cycuity Radix fill gaps in the design process that Cadence does not cover. This perspective was valuable to my research because it helped me understand that using one vendor's products doesn't mean ignoring the rest of the field, but how it's important to be aware of how different tools complement or compete with each other, especially when working on something as interdisciplinary as a RISC-V security extension. By reading through these different attributes, and comparing them in our research section, I was able to gain more insight into more industry products than just Cadence.

Overall, ChatGPT helped guide my research in a structured way while still letting me control the direction. Each refined prompt produced clearer, more tailored information, avoiding the inconclusive and time consuming search if I had used regular search engines. It also allowed me to build a strong conceptual map of the EDA ecosystem, understand where my project fits within industry practices, and feel more confident moving forward with Cadence tools. This early exploration ultimately strengthened the foundation of the project and helped me approach the design work with a much more informed perspective.

Chapter 6 - Hardware & Software Design

This chapter documents the hardware and software design of our RISC-V subsystem built around the Ibex core[1]. 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 so 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 to enable the end-goal functionality, and (3) to connect 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 Section 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.

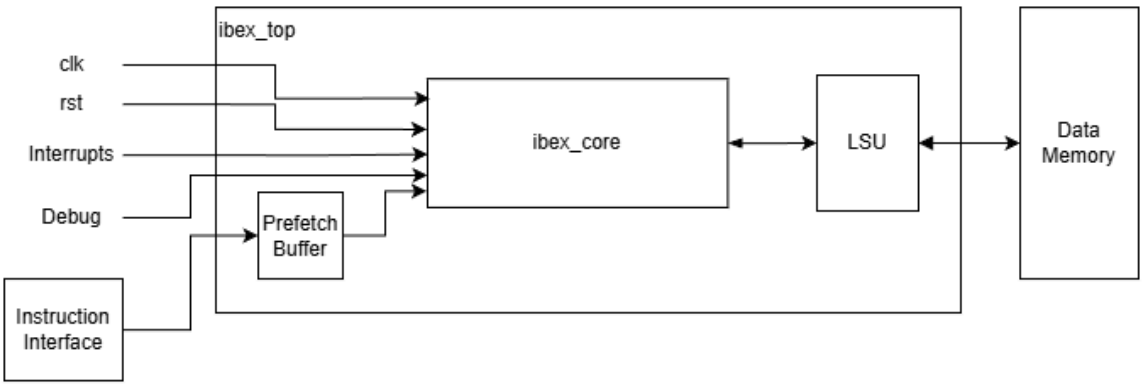


Figure 6.1: *Ibex Top Module*

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 in verification environments, it is not intended to be synthesized in hardware.

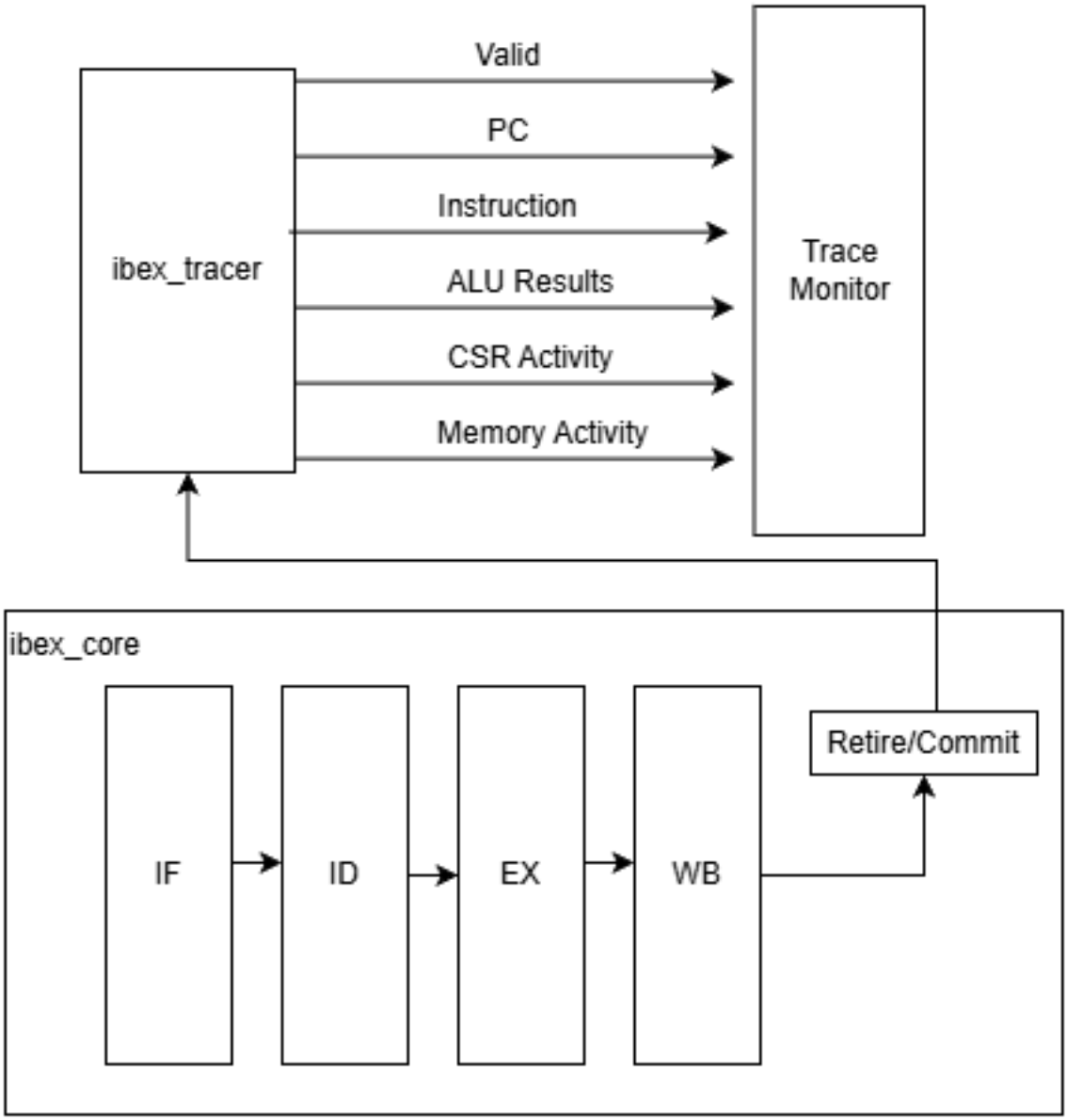


Figure 6.2: *Ibex Top Tracing Module*

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.

Table 6.1: *Definitions Made in ibex_pkg*

Enum	Included In	Purpose
crash_dump_t	ibex_top_tracing, ibex_lockstep	Captures architectural state for crash/debug reporting, including the current PC, next PC, last accessed data address, and exception-related PCs/addresses. Used during fatal exceptions and error escalation to report processor state.
regfile_e	ibex_core , ibex_register_file_ff, ibex_top	Selects which register file implementation is used: flip-flops, FPGA block RAM, or latch-based design. Controls which RF module is instantiated.
rv32m_e	ibex_core, ibex_ex_block, ibex_decoder, ibex_multdiv_slow, ibex_multdiv_fast	Configures hardware multiply/divide support: none, slow iterative, fast multicycle, or single-cycle M extension unit. Changes pipeline behavior and EX-stage datapath.
rv32b_e	ibex_core, ibex_alu, ibex_decoder, ibex_id_stage	Controls support level for the RISC-V Bit-Manipulation extension: none, balanced subset, Earl Grey optimized subset, or full B-extension feature set. Affects ALU submodule generation.
alu_op_e	ibex_decoder, ibex_ex_block, ibex_alu, ibex_id_stage	Defines all ALU operation codes (arithmetic, logical, shifts, comparisons).
md_op_e	ibex_multdiv_slow, ibex_multdiv_fast, ibex_ex_block, ibex_decoder	Selects the specific M-extension arithmetic operation: low multiply (MULL), high multiply (MULH family via additional decode bits), division (DIV), or remainder (REM). Drives the mult/div unit's internal operation selection.
csr_op_e	ibex_id_stage, ibex_cs_registers	Defines the CSR instruction operation: read (CSRR), write (CSRRW), bit-set (CSRRS), and bit-clear (CSRRC). Used by decode and CSR register file control logic.
priv_lvl_e	ibex_cs_registers, ibex_pmp	Represents RISC-V privilege modes: Machine, Hypervisor (reserved), Supervisor, and User. Used for CSR access control, trap routing, and PMP permission checks.
x_debug_ver_e	ibex_debug_module, ibex_cs_registers	Encodes which external debug architecture is implemented: none, standard RISC-V debug spec, or a non-standard debug interface. Used when constructing the DCSR CSR.

Continued on next page

Enum	Included In	Purpose
wb_instr_type_e	ibex_wb_stage, ibex_core	Indicates the type of instruction being written back: load (waiting for data), store (waiting for store response), or other (ALU, CSR, or non-memory instructions). Used to control writeback stage and track pending memory operations.
op_a_sel_e, op_b_sel_e	ibex_decoder, ibex_id_stage, ibex_ex_block	Selects operand A and B source (rs1, rs2, PC, immediate).
imm_a_sel_e, imm_b_sel_e	ibex_decoder, ibex_id_stage	Chooses which immediate bits are used for instruction decode.
opcode_e	ibex_decoder, ibex_branch_predictor	Defines the primary instruction-class opcodes (LOAD, STORE, OP-IMM, OP, LUI, AUIPC, BRANCH, JAL, JALR, SYSTEM). Used by the decoder to identify the instruction type.
rf_wd_sel_e	ibex_decoder, ibex_id_stage	Selects the source for register file writeback data: from the EX-stage ALU result or from a CSR operation result.
ctrl_fsm_e	ibex_controller	Defines the main pipeline controller FSM states: RESET, BOOT_SET, WAIT_SLEEP, SLEEP, FIRST_FETCH, DECODE, FLUSH, IRQ_TAKEN, DBG_TAKEN_IF, DBG_TAKEN_ID. Controls stalls, flushes, exceptions, and debug entry.
pc_sel_e	ibex_if_stage, ibex_controller	Controls the source of the next PC in the IF stage: boot vector, jump target, exception vector, ERET, DRET, or branch predictor target.
exc_pc_sel_e	ibex_if_stage, ibex_controller	Selects which PC to use during exception handling: exception entry, IRQ, debug break, or exception while in debug mode. Used by the IF stage and controller to redirect fetch.

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.

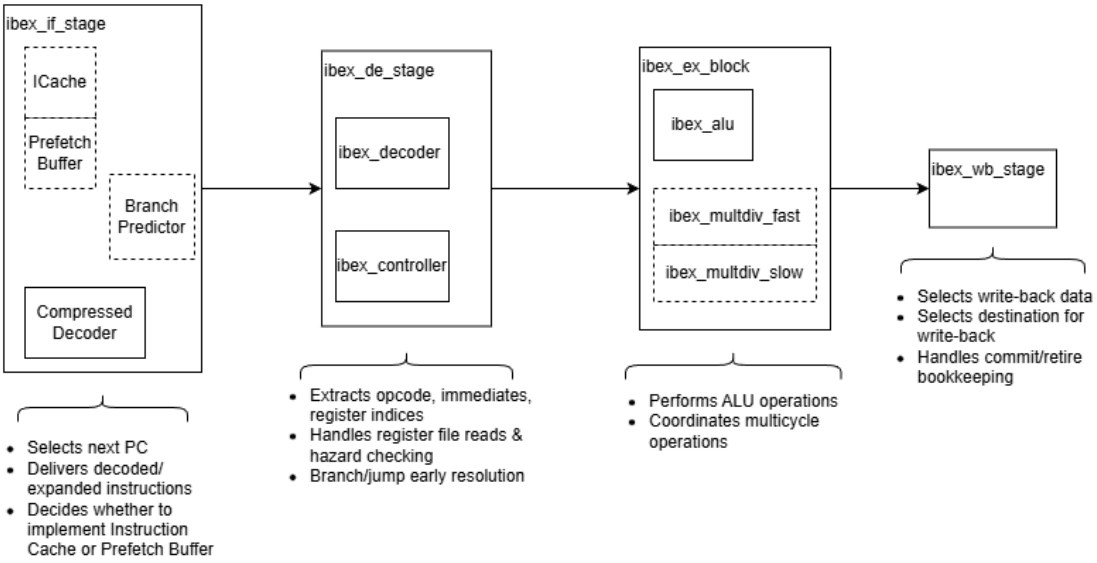


Figure 6.3: *Ibex Pipeline Stages*

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.

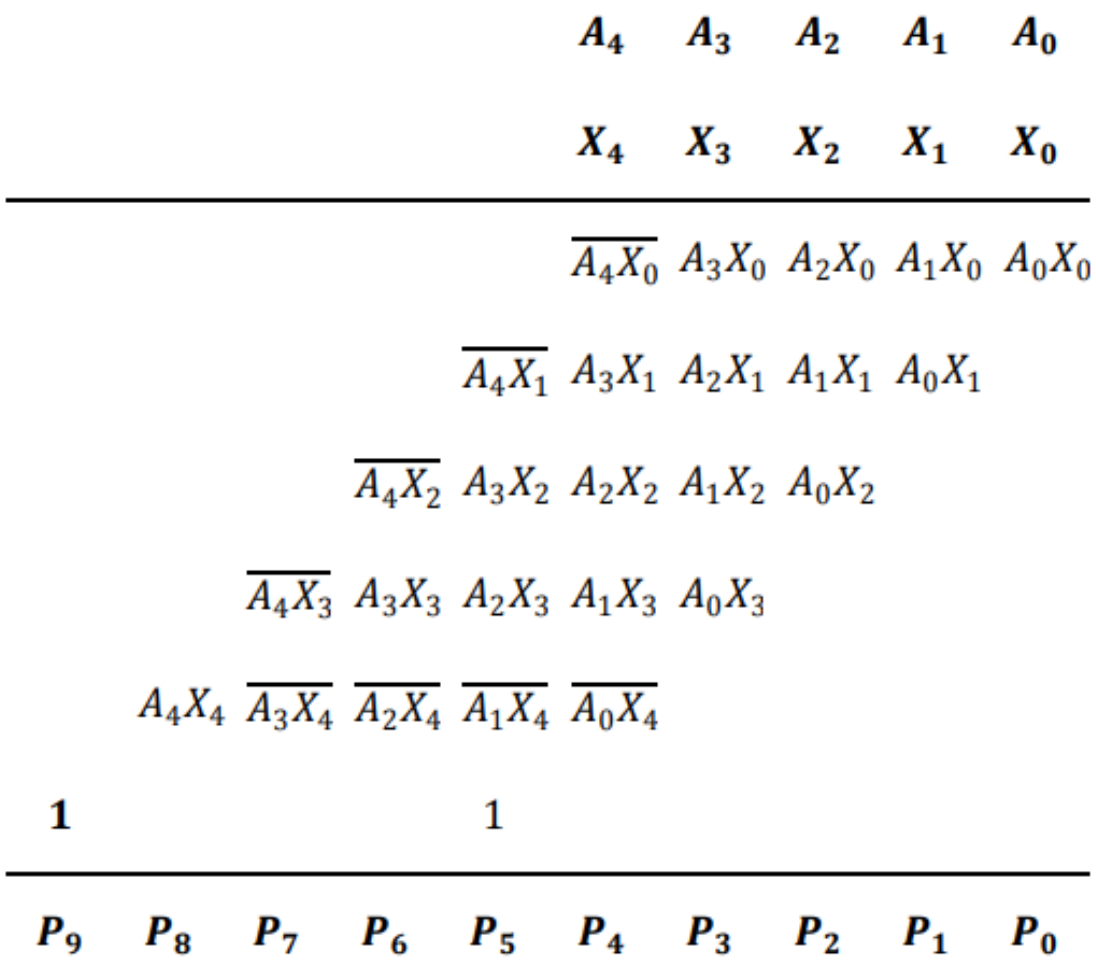


Figure 6.4: *5-bit Baugh-Wooley Multiplication*

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

The 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 exception 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 is 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 with inference of hardware carry chains/DSP blocks on some FPGAs, improving Fmax. Width is parameterized; in Ibex, counters can be 40+ bits internally and can be 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.

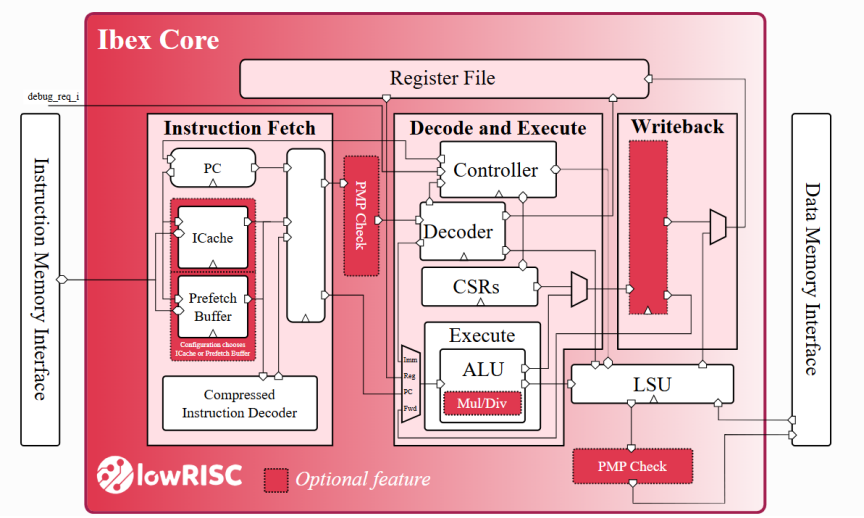


Figure 6.5: *Diagram of Ibex Core [1]*

6.6 Core Redesign for Security

6.6.1 Overview

Now the functionality of the base core has been covered, this section will focus on additions made to the hardware to increase mitigation of Side-Channel Attacks. Below is

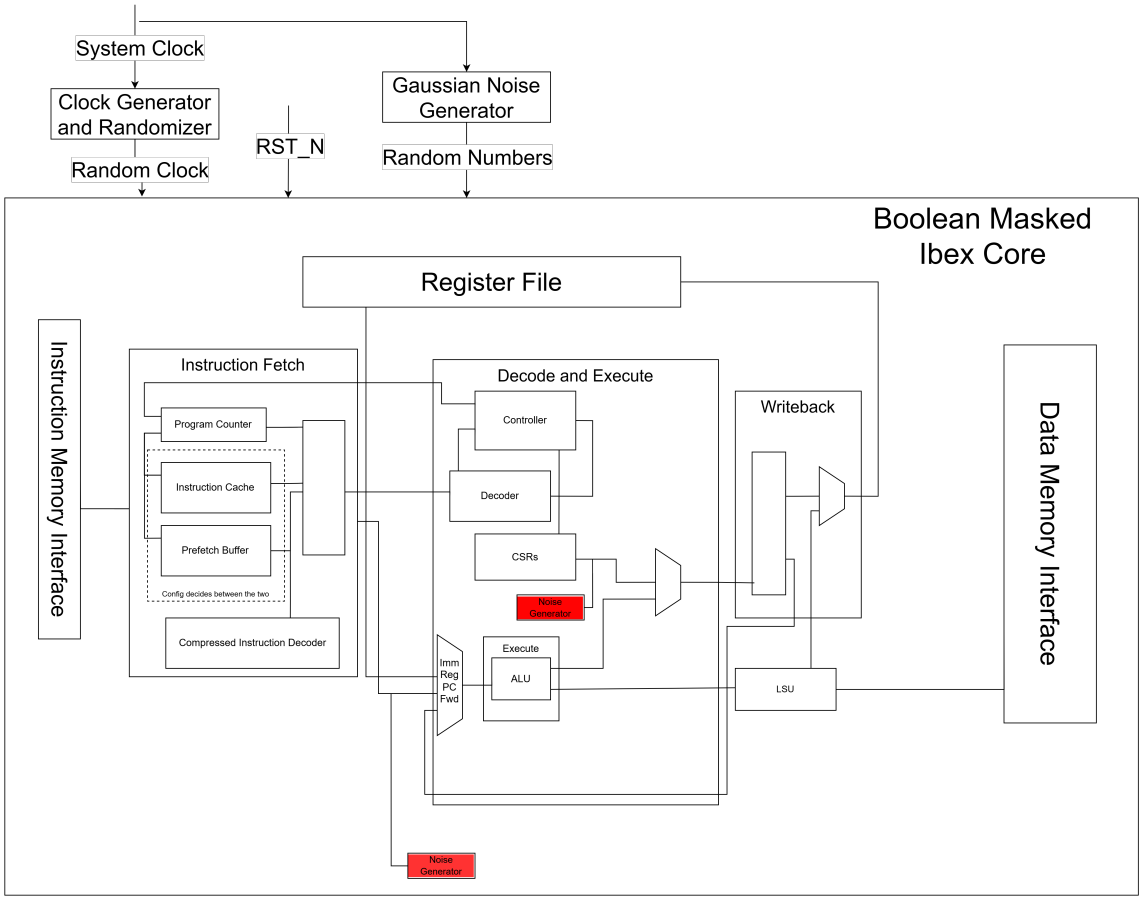


Figure 6.6: *Schematic of Redesign of Ibex Core*

a schematic of the redesigned core which shows the addition of the clock generator and randomizer, random number generators, noise generators and the portion of the core that is Boolean Masked.

These additional modules were developed in Verilog to ensure seamless integration with the existing processor design. The custom clock randomization system is based on concepts from prior work but was redesigned to eliminate dependence on ring oscillator jitter. In the proposed implementation, an 8 bit LFSR governs the skipping of clock edges. The two least significant bits determine when a skip occurs, and a multiplexer allows the selection of skip rates of 0 percent, 25 percent, 50 percent, or 75 percent. In addition, noise generation modules were implemented that activate in response to internal enable signals within the core. Both the randomized clocking mechanism and the noise generators are categorized as hiding countermeasures, as they focus on obfuscating the observable power traces and increasing the difficulty of extracting meaningful information from them. The final technique, Boolean Masking, also falls within the domain of hiding countermeasures, but it operates directly on the data being processed. This is achieved by replacing standard logic cells with secure masked variants, ensuring that internal signal transitions are less correlated with sensitive values and therefore more resistant to side-channel analysis. Considerations and requirements related each technique used in the redesign are detailed in later sections.

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 data into shares that are independent from itself. 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, it's critical to ensure the operations done one the shares preserve the original data. Essentially, it is important that the outputs of these operations can be used to reconstruct the expected result. Linear operations such as XOR, shifts, and permutations do not require modification and are applied to both shares. For non-linear operations, this is challenging and requires care to make sure that data is not leaked. Thus, the standard AND and OR gates must be converted into an equivalent masking/secure gate that takes in the two shares as input. This project opted to use the masking method presented by Biruykov [39] as it reduces the amount of random numbers needed 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.

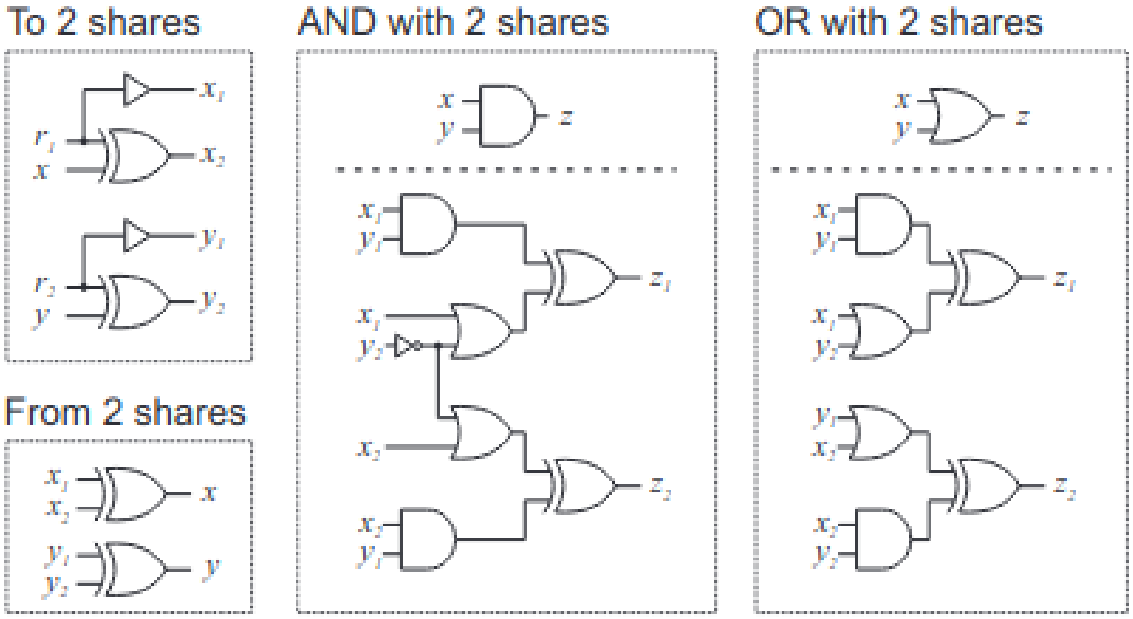


Figure 6.7: *Masking versions for the non-linear gates*

Since the gates above are not available in the current PDK, building the gates required using structural Verilog to instantiate and connect gate primitives present in the GSCLib045 such that it would match the schematic. Implementing these new masked gates can be accomplished by updating the 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 and open new leakages in the circuit. Work done by Stangherlin [24] has provided an outline for these Tcl scripts, which will be used as a framework for this project.

After setting up the environment (reading libraries from PDK and setting up paths), 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 PRNG chosen for Boolean Masking was a Gaussian White Noise Generator (GNG) which is a Verilog IP Cores from OpenCores. This IP Core was chosen over other PRNGs as Boolean Masking requires fresh new random bits to be effective. In order to accomplish this with PRNGs, generators with longer periods are preferred and this GNG provides a period of 2^{176}. Since the core is open-source, all that needs to be done is to change the name of the output signal that contains the generated random number to randbits so that it can be used by the tcl script. The top module design for the GNG is below and more details can be found in the associated repository documentation[34].

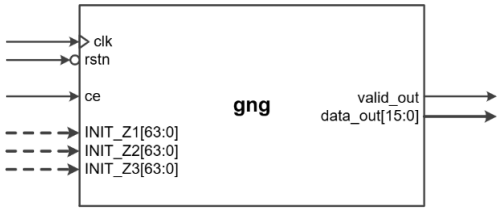


Figure 6.8: *Core schematic*

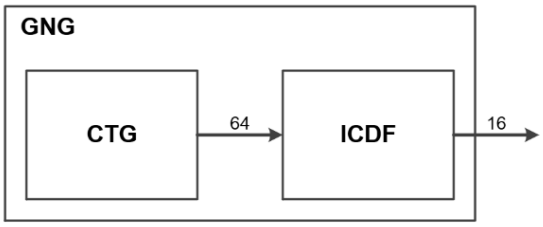


Figure 6.9: *Top Module Architecture*

After this, 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 for mitigating side-channel attacks, where adversaries attempt to extract confidential information by analyzing power traces. For this project, a ring oscillator and a Cellular Automata Shift Register are used where they switch on based on an enable signal, which is driven by an XOR combination of a set of select lines so that noise generators will turn on given certain modules being on.

6.6.3.1 Ring Oscillator

As previously stated, the purpose of the ring oscillator is to generate noise in the circuit during certain operations. This is accomplished by using an enable signal to

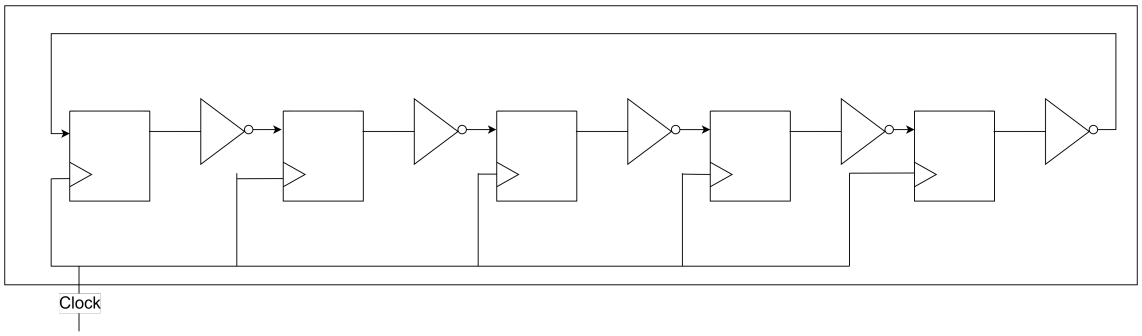


Figure 6.10: *Schematic of Ring Oscillator*

The choice to utilize circuits with high switching activity is deliberate. Switched circuits frequently exhibit increased dynamic power dissipation, which in turn produces larger and less predictable fluctuations in 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.

Table 6.2: *Signals in ring_oscillator Module*

Signal	Input/Output/Internal	Bit-Length	Purpose
clk	Input	1	Clock used for synchronous progression of the ring oscillator logic.
enable	Input	1	Enables the ring oscillator. When HIGH, the stages begin toggling to generate oscillation.
rst_n	Input	1	Active-low synchronous reset. When LOW, stages are reset to 0.
stages	Internal	5	Register array representing the chain of inverting stages that form the ring oscillator. Each bit toggles based on the previous stage.
i	Internal	NA	Integer loop variable used to iterate through the stage assignments inside the always block.
unused	Internal	1	Reduction OR of all stage bits. Prevents synthesis tools from optimizing away the oscillator stages.

6.6.4 Cellular Automata Shift Register

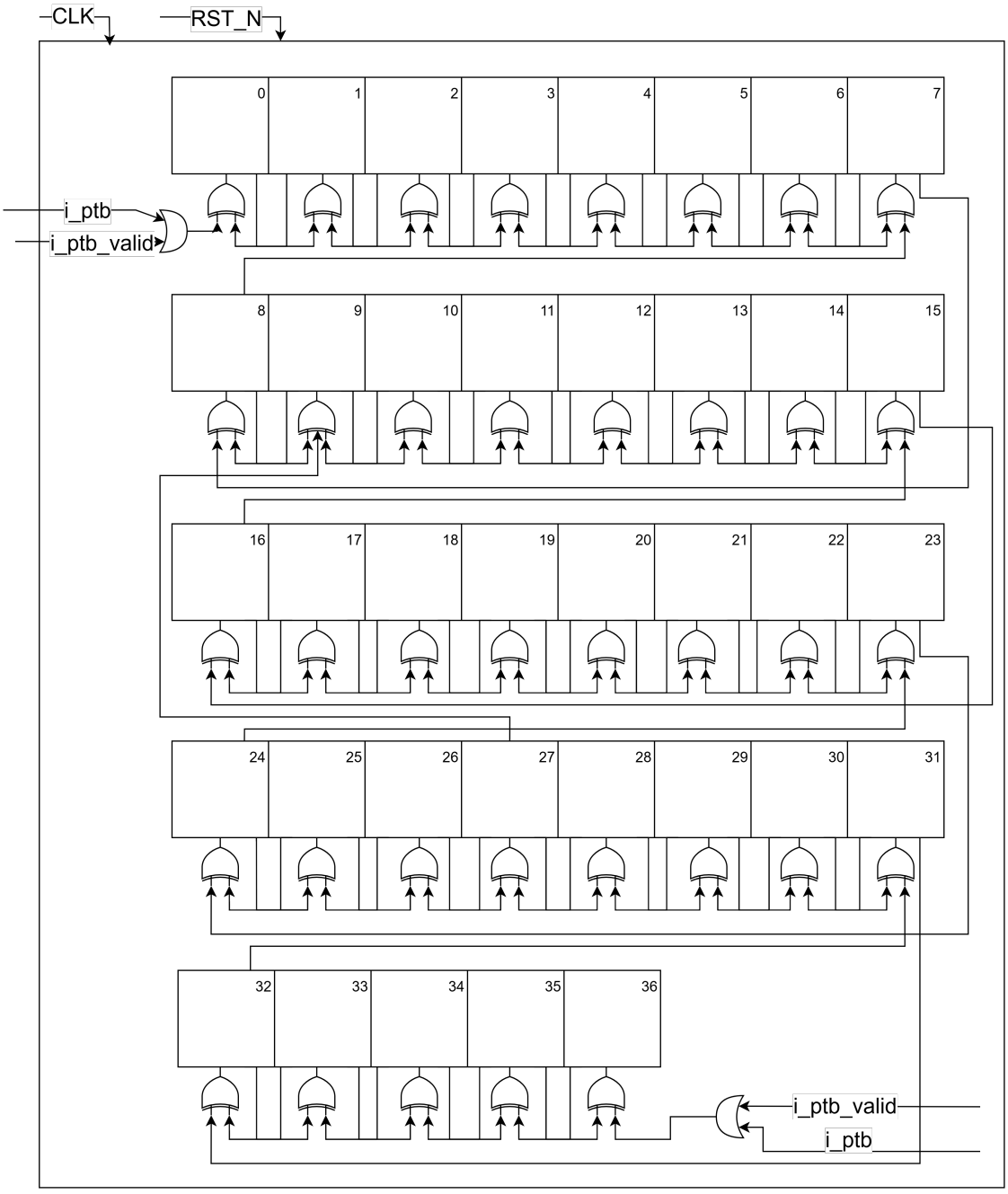


Figure 6.11: *Schematic of 37-bit CASR*

A cellular automata shift register (CASR) is a form of shift register that evolves according to the rules of a one-dimensional cellular automaton. Each bit, or cell, updates its value based on the values of its neighboring cells rather than shifting uniformly in one direction like a conventional linear feedback shift register. Since the update behavior is distributed and depends only on local interactions, CASRs can generate complex and

often pseudo-random sequences. This makes them useful in applications such as random number generation, lightweight cryptography, and hardware systems that benefit from simple, uniform update logic.

The Verilog module implements a CASR with thirty-seven cells stored in the output register. On each clock cycle, if the enable signal is active, every bit in the register is updated simultaneously according to a predefined rule. The module uses a reversed indexing convention to map loop indices to bit positions, but the essential behavior is that each cell looks at its left and right neighbors to determine its next state. For almost all positions, the rule computes the next value as the exclusive-or of those two neighboring bits. This corresponds to a well-known cellular automaton rule that produces evolving patterns with good pseudo-random characteristics.

One position in the register, cell nine, uses a slightly modified update rule. Instead of depending only on its neighbors, it also incorporates its own current value into the update. This small alteration helps break up short repeating cycles and improves the overall sequence quality produced by the register. In addition, the two boundary cells cannot take values from neighbors that do not exist, so the design allows an external perturbation bit to be injected when valid. This feature prevents the register from falling into trivial or stagnant states by providing occasional external influence.

When the reset signal is asserted, the entire register clears to zero, and when the enable signal is inactive, the state simply holds. Overall, the module implements a compact cellular automata shift register with a special tap and optional perturbation to improve randomness and robustness.

Table 6.3: *Signals in* `casr37` *Module*

Signal	Input/Output/Internal	Bit-Length	Purpose
clk	Input	1	Clock signal used to synchronously update the cellular automata state.
rst_n	Input	1	Active-low reset that clears the entire 37-bit state to zero.
i_en	Input	1	Enable signal that allows the CASR to evolve. When LOW, the state holds its previous value.
i_ptb	Input	1	Perturbation bit injected into the boundary cells to prevent the automaton from entering trivial or stagnant states.
i_ptb_valid	Input	1	Indicates when the perturbation bit is valid and should replace boundary neighbor values.
o_state	Output	37	The 37-bit register representing the current state of the cellular automata shift register.
nxt_state	Internal	37	Combinational next-state values for all cells, computed from neighbor logic and special rules.
idx	Internal	–	Integer loop variable used in the sequential block to copy next-state values into the state register.
i	Internal	–	Generate-loop variable used to instantiate repeated next-state logic for each of the 37 cells.
left	Internal	1	For each generated cell, the left neighbor bit or the perturbation bit when evaluating a boundary cell.
right	Internal	1	For each generated cell, the right neighbor bit or the perturbation bit when evaluating a boundary cell.

6.6.5 Clock Generator and Randomizer

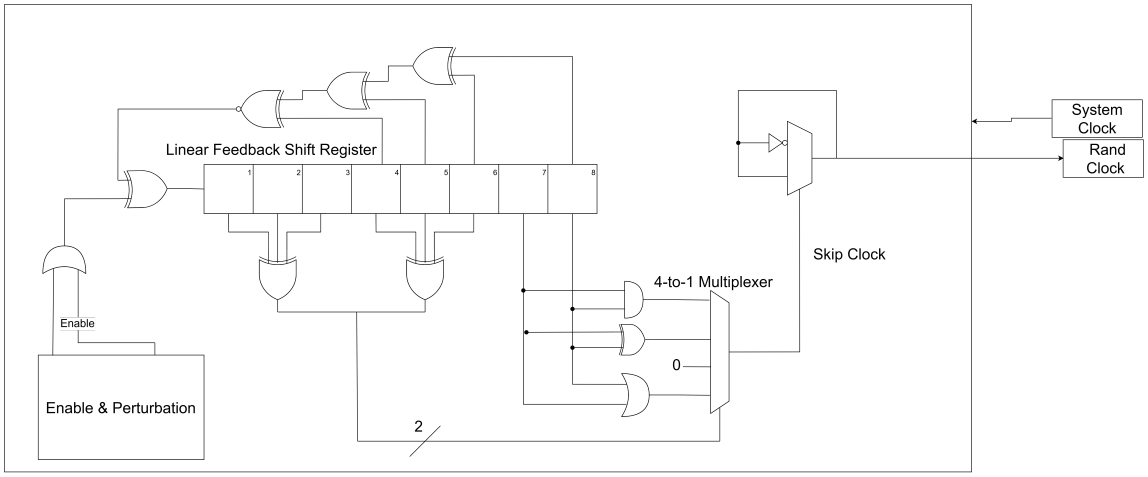


Figure 6.12: *Schematic of Clock Generator and Randomizer*

6.6.5.1 Summary

The clock randomization technique implemented in this project is conceptually similar to the method presented in [24], but with several key modifications to create custom Clock Generator and Randomization Module. Unlike the referenced design, this project cannot rely on jitter derived from a ring oscillator to produce truly random values. Instead, the output clock is randomized using the two least significant bits (MSBs) of an 8-bit linear feedback shift register (LFSR), which determine when a clock edge should be skipped. A multiplexer was introduced to control the skip frequency, allowing selectable skip rates of 25%, 50%, and 75% corresponding to the logical AND, XOR, and OR outputs, respectively. An additional mode in which no clock edges are skipped (output tied to zero) is also provided.

To dynamically choose among these skip rates, the three logic outputs are fed into a multiplexer controlled by a 2-bit select signal, derived from the six most significant bits (MSBs) of the same 8-bit LFSR. Furthermore, to enhance randomness and reduce periodicity, a perturbation mechanism was added to the LFSR. Every 32 cycles, a perturbation bit is generated through an XOR-reduction of a subset of registers within a separate 43-bit LFSR. This bit is then used to modify the original 8-bit LFSR seed, introducing additional variability and improving the unpredictability of the generated clock behavior.

Table 6.4: *Table of Signals in Clock Generator and Randomizer Module*

Signal	Input/Output/Internal	BitLength	Purpose
i_clk	Input	1	System Clock goes here
rst_n	Input	1	Asynchronous Reset, when set to low it initializes the CGR
o_clk	Output	1	Output of the CGR. This is the random clock that is used to drive the processor
lfsrOutput8	Internal	8	8-bit value that represents the current value being stored in the LFSR
selectLines	Internal	2	Select line used to control a 4-to-1 mux that selects how often the clock edges are getting skipped
p25	Internal	1	Wire that contains the output from an AND from the MSB of the LFSR
p50	Internal	1	Wire that contains the output from an XOR from the MSB of the LFSR
p75	Internal	1	Wire that contains the output from an OR from the MSB of the LFSR
skip_clk	Internal	1	Output selected by the mux
skip_clk_r	Internal	1	This is the latched output of the skip_clk
ptb_bit	Internal	1	Bit used to perturb the 8-bit LFSR that is generated by doing an XOR Reduction of a subset of register of the 43-bit LFSR
ptb_bit_valid	Internal	1	Bit is set to HIGH once every 32 cycles that is the enable for ptb_bit to affect the 8-bit LFSR

6.6.5.2 Control Logic for Random Clock

Select Line:	Signal related to select signal:	Value assigned to signal:	Result:
00	N/A (Value is just tied to zero)	1'b0	Does not skip any clock edges
01	p25	(lfsrOutput8[7] & lfsrOutput8[6])	Skips 25% of clock edges
10	p50	(lfsrOutput8[7] ^ lfsrOutput8[6])	Skips 50% of clock edges
11	p75	(lfsrOutput8[7] | lfsrOutput8[6])	Skips 75% of clock edges

When the reset logic occurs the clock is initialized to low. The random clock is controlled by a 4-to-1 mux whose values and select lines are generated by operations done on the registers of the LFSR at every rising edge of the system-clock. The two bit select line for the mux is the concatenation of the XOR reduction of the bit range 1-3 and 4-6. Whenever the value 1 is read from the mux during the rising edge the clock edge is skipped and the value of the clock does not change. The values the select lines choose from is an output of zero (representing default where no clock edges get skipped) or the output of an AND (25% of edges get skipped), OR (75% of edges get skipped), or XOR (50% of edges get skipped) of the 2 most significant bits of the LFSR.

6.6.5.3 Perturbation Logic

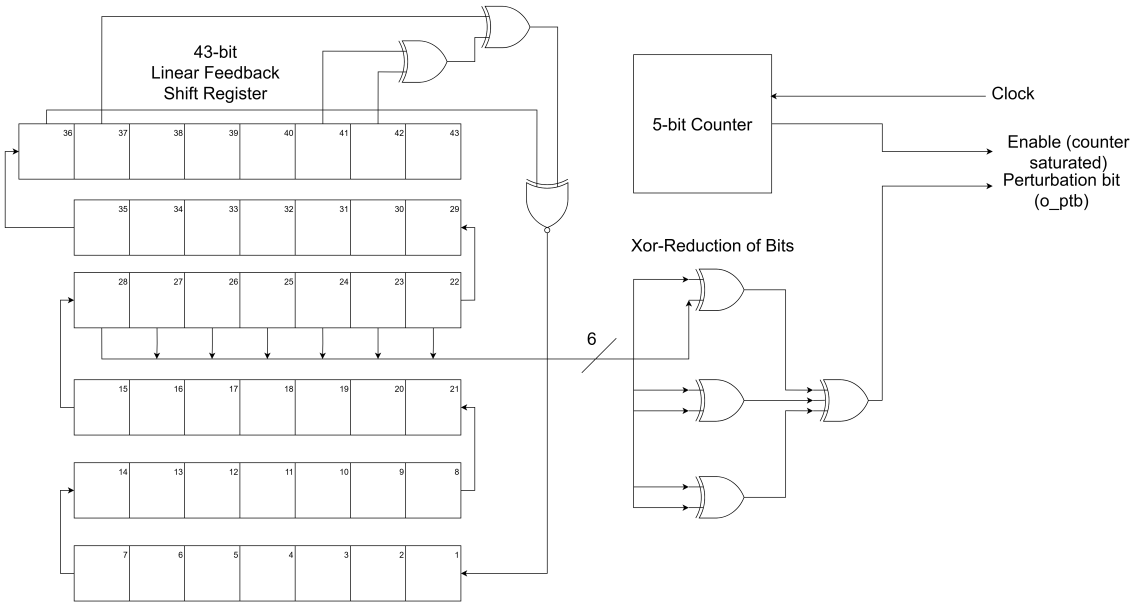


Figure 6.13: *Schematic of Perturbation Module*

Every 32 cycles the feedback loop of the LFSR is perturbed by a bit generated from the XOR reduction of a subset of registers in a 43-bit LFSR. Every clock cycle the 43-bit LFSR generates a perturbation bit (`o_ptb`), but it's only every 32 cycles when a `o_ptb_valid` bit is set to 1 that allows for `o_ptb` to effect the feedback loop of the 8-bit LFSR. A 5-bit counter is used to determine when the perturbation bit is valid. Once the counter reaches 32 cycles it turns on the `ptb_valid` bit for one cycle and then set it back to zero. The schematic for this module is shown in above for a more visual representation.

Table 6.5: *Table of signals for Perturbation Logic*

Signal	Input/Output/Internal	Bit-Length	Purpose
clk	Input	1	Port for clock signal
rst_n	Input	1	Asynchronous active-low reset
o_ptb	Output	1	Bit used to perturb generated by the 43-bit LFSR to perturb the 8-bit LFSR
o_ptb_valid	Output	1	Enable for o_ptb to perturb the 8-bit LFSR
feedback	Internal Wire	1	Stores the value to be shifted into the LFSR based on the XNOR of register 42,41,37,36 in the 43-bit LFSR
max_pulse_counter	Internal Wire	1	This is the enable to switch o_ptb_valid. Goes HIGH for one cycle every 32 cycles based on if a 32-bit counter
o_state	Internal Register	43	43-bit register that stores the value of the bits for the LFSR

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.

6.7.2 Verification of Custom Sub-modules

6.7.2.1 casr37.v Verification Process

The `casr37` module implements a 37-bit cellular automata shift register (CASR). Each bit of the next state is computed as the XOR of its left and right neighbors, with a special rule at cell 9 that also includes the current bit value. At the two ends of the register, the neighborhood is terminated by the external perturbation input `i_ptb` when `i_ptb_valid` is asserted. An enable input `i_en` gates state updates so that the CASR can be held when required. The verification goal is to confirm correct reset behavior, correct gating through `i_en`, and correct neighbor-based evolution with and without perturbations from `i_ptb`.

The testbench for `casr37` generates a 100 MHz clock, applies an active-low reset, and then drives the control and perturbation inputs in several phases. At the start of simulation, `rst_n` is held low while all other inputs are zero. After reset is released, `i_en` remains low for several cycles to confirm that the output state `o_state[36:0]` holds at all zeros even as the clock toggles. `i_en` is then asserted and a short burst of perturbations is applied at the boundaries by driving `i_ptb = 1` with `i_ptb_valid = 1`. Later in the run, `i_en` is deasserted again while `i_ptb` and `i_ptb_valid` toggle, and finally reasserted with `i_ptb` driven in a more irregular pattern to exercise the perturbed evolution of the CASR.

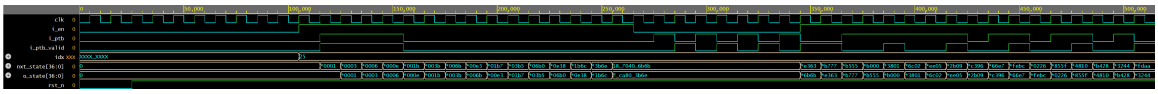


Figure 6.14: *`casr37.v` testbench waveform showing reset, enable gating, and CASR evolution with and without perturbations.*

As shown in Figure 6.14, while `rst_n` is low, `o_state` is held at `0`. After reset deassertion and with `i_en` still low, the state remains zero across multiple clock cycles, confirming that the enable correctly prevents any updates. Once `i_en` is asserted and a perturbation is applied (with `i_ptb` and `i_ptb_valid` both high for a few cycles), the next-state vector `nxt_state[36:0]` quickly becomes non-zero, and on the following clock edges the registered state `o_state` follows it. Early in the enabled phase, the waveform shows a sequence such as

$$\texttt{o\_state}: \; 0000000\ldots0 \rightarrow \texttt{0001} \rightarrow \texttt{0003} \rightarrow \texttt{0006} \rightarrow \texttt{000C} \rightarrow \texttt{001B} \rightarrow \texttt{003B} \rightarrow \ldots$$

which is consistent with the neighbor-based update rule and with `o_state` lagging the purely combinational `nxt_state` by one clock cycle.

In the mid-simulation hold phase, the waveform shows that `o_state` remains constant across several clock cycles even though `i_ptb` and `i_ptb_valid` continue to toggle, confirming that the enable gate is effective. When `i_en` is reasserted with irregular perturbations on `i_ptb`, the CASR produces a richer sequence of non-zero patterns, and `o_state` continues to

Table 6.6: *Observed behavior of key signals in the* `casr37` *waveform.*

Signal	Observed behavior
`clk`	Free-running 100 MHz clock.
`rst_n`	Active-low reset that forces `o_state` to all zeros.
`i_en`	When low, holds `o_state` constant; when high, updates `o_state` on each rising edge of `clk`.
`i_ptb`	External perturbation bit applied at both ends of the CASR.
`i_ptb_valid`	Qualifies `i_ptb`; when low, the edges behave as null-terminated (no perturbation).
`nxt_state[36:0]`	Combinational next-state vector derived from left/right neighbors and the special rule at cell 9.
`o_state[36:0]`	Registered CASR state; equals the previous cycle's `nxt_state` when `i_en` is high and holds its value when `i_en` is low.

track `nxt_state` with the expected one-cycle delay. Throughout the entire run, no unknown or unstable values are observed, and the behavior matches the intended neighbor-based CASR model with optional edge perturbation.

6.7.2.2 clkgen.v Verification Process

The clkgen.v module, as previously described in the design modules, implements the randomized clock generator whose purpose is to misalign the power traces and disrupt timing-based side-channel attacks. It consumes a free-running system clock and conditionally suppresses clock edges based on the pseudo-random control logic derived from an internally-instantiated 8-bit LFSR Additional entropy is also introduced by periodically perturbing the 8-bit LFSR once every 32 cycles. The goal of verifying this module is to confirm that the clock initializes deterministically on reset, the skip decisions correspond exactly to their respective multiplexed behavior, the perturbation affects the 8-bit LFSR only during valid windows, and that `o_clk` toggles on cycles where skipping is disabled.

The SystemVerilog testbench will instantiate clkgen and produce a 100MHz clock in addition to an active low reset. At time zero, `rst_n` is held low and all internal signals are driven to known values. After several rising edges, reset is released and the randomized clock generation begins. The stimulus sequence is run in phases: first observing behavior with deterministic LFSR output, then running long sequences through multiple 32-cycle windows where `o_pt_valid` pulses periodically. Then randomized enable conditions are applied and with constrained random delay windows to thoroughly test different skip patterns.

6.7.2.3 lfsr43.v Verification Process

This section outlines the key behaviors of the lfsr43 module that were targeted for verification, the testbench environment used to validate these behaviors, and the expected simulation results. The goal of this modules verification is to ensure that the LFSR, output-bit generation logic, and the 5-bit insertion-counter all function correctly under reset and normal operating conditions.

Verification was performed using a compact behavioral testbench that provides a controlled and realistic simulation environment. The testbench instantiates the lfsr43 DUT, generates a free-running 100MHz clock (10 ns period), applies an active-low reset sequence, and then observes the module's outputs over an extended period of time. During initialization, `rst_n` is held low to confirm that all internal registers respond correctly to reset, then after 20 ns the reset is released, and the LFSR and counter begin updating on each rising clock edge. The simulation then continues for 200 additional clock cycles, during which the LFSR state, output bit (`o_ptb`), and valid-pulse signal (`o_ptb_valid`) are observed.

Waveform generation is enabled using $dumpfile and $dumpvars, allowing all relevant signals to be captured for analysis. The test ends with $finish once the observation window has been completed.

The first functional behavior examined was reset initialization. When the active-low reset signal (`rst_n`) is asserted, the 43-bit LFSR must load the fixed non-zero seed value 43'h1ABCDE12345, while both output signals (`o_ptb` and `o_ptb_valid`) and the internal 5-bit counter must return to zero. Simulation confirmed that all registers entered their specified reset values immediately, no undefined or metastable states were generated, and the module transitioned cleanly into normal operation once the reset was deasserted.

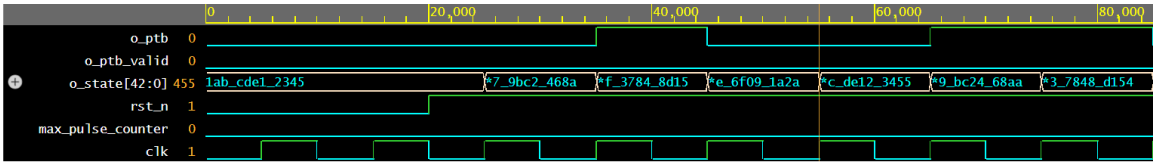


Figure 6.15: *lfsr43.v Testbench Waveform*

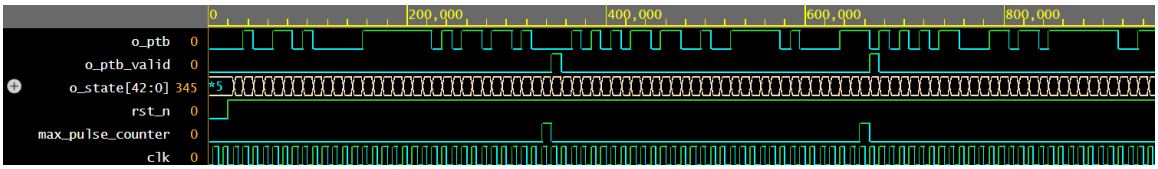


Figure 6.16: *lfsr43.v Testbench Waveform (zoomed out)*

In the waveform, the LFSR shift behavior is clearly demonstrated through consecutive state values such as 357_9bc2_468a, 6af_3784_8d15, and 55e_6f09_1a2a. Each new value is the previous state shifted left by one bit, with the least-significant bit replaced by the computed feedback value. For example, shifting 357_9bc2_468a yields 6af_3784_8d1X, and the waveform shows the inserted feedback bit to be 1, producing 6af_3784_8d15. The same pattern occurs in the next transition to 55e_6f09_1a2a. This confirms that both the left-shift operation and feedback insertion logic are functioning exactly as intended.

The output bit `o_ptb` was also verified. This signal is intended to represent the XOR-reduction of LFSR bits [27:21]. During simulation, the testbench captured both the selected state slice and the output, and manual comparison confirmed that `o_ptb` consistently matched the XOR of the seven selected LFSR bits. This provides confidence that the pseudo-random output is being generated correctly and without timing glitches.

Another essential element of verification involved the 5-bit counter, which drives the `o_ptb_valid` pulse. The counter is required to increment monotonically from 0 to 31 and

Table 6.7: *Observed behavior of key signals in the `lfsr43` waveform*

Signal	Observed Behavior
`clk`	Free-running clock; toggles every fixed time step.
`rst_n`	Held low at start, then driven high to release reset.
`o_state[42:0]`	Initializes to seed `43'h1ABCDE12345` after reset and then shifts left by 1 bit every clock with a new feedback bit inserted; hexadecimal values appear pseudo-random over time.
`o_ptb`	Single-bit output derived as XOR of `o_state[27:21]`; toggles in a pseudo-random pattern aligned to clock edges.
`max_pulse_counter`	Internal pulse from 5-bit counter; asserted for one cycle when the counter reaches its terminal count.
`o_ptb_valid`	Mirrors `max_pulse_counter`; produces a single-cycle high pulse every 32 clock cycles indicating a "valid" output bit.

assert its max-pulse output only when reaching the terminal count. This pulse is reflected in the LFSR module as the signal `o_ptb_valid`. By monitoring the counter and valid-pulse output across simulation, it was observed that a clean, single-cycle pulse occurred exactly every 32 cycles, and that the counter correctly wrapped back to zero after reaching 31. This confirmed the proper operation of the spacing mechanism used to mark periodic sampling opportunities.

To verify the correctness of the output-bit generation logic, I extracted the seven bits `o_state`[27:21] from the waveform and computed their XOR-reduction manually. For example, when the slice was 1000010, the XOR of all seven bits evaluates to 0, which exactly matched the value of `o_ptb` for that cycle. On the following cycle the slice transitioned to 0000100, whose XOR reduces to 1, and again `o_ptb` reflected this value. Repeating this for multiple cycles (including slices such as 0000100 1 and 0100001 0) consistently produced a perfect match. Because the mathematically computed reduction matches the hardware output across several consecutive cycles, this confirms that `o_ptb` is being generated correctly from the XOR-reduction of `o_state`[27:21], and the signal exhibits the expected pseudo-random toggling behavior.

Slice *o_state*[27:21]	**XOR Result**	**Expected** *o_ptb*
1000010	0	0
0000100	1	1
0000100	1	1
0100001	0	0

Figure 6.17: *Verification of o_ptb through XOR-reduction of the LFSR bits o_state*[27:21]*. Each row corresponds to one clock cycle from the waveform. The computed XOR value matches the RTL output o_ptb for all cycles, confirming correct implementation of the expression o_ptb* = ^*o_state*[27:21].

The testbench also ensured that the LFSR state updates continued normally even

when the valid pulse occurred. Overlaying the LFSR state, output bit, and pulse signal in the waveform viewer demonstrated that the counter logic did not interfere with LFSR progression, and both subsystems operated concurrently as intended. Finally, a check was performed to ensure that the 43-bit state never reached the illegal all-zero condition. The simulation window showed continuous non-zero states throughout execution, indicating that the maximal-length LFSR structure remained valid and stable.

Overall, simulation results indicate that the module behaves as intended. Reset logic functions correctly, the LFSR produces valid pseudo-random sequences, the output bit is derived properly, the counter generates clean periodic pulses, and no illegal or unstable states are entered. These tests collectively verify the correctness and robustness of the lfsr43 module

6.7.2.4 lfsr8.v Verification Process

The `lfsr8` module implements an 8-bit XNOR-based linear-feedback shift register (LFSR) with an optional external mixing bit. The verification goal is to confirm that the LFSR initializes to the correct non-zero seed, advances according to the polynomial $x^8 + x^6 + x^5 + x^4 + 1$, and correctly incorporates the external bit `i_ptb` when `i_ptb_valid` is asserted, while never entering the all-zero state.

A compact behavioral testbench instantiates the DUT, generates a free-running 100 MHz clock (10 ns period), applies an active-low reset, and then drives several phases of stimulus. At time zero, `rst_n` is held low and all other inputs are driven to zero. After a few rising clock edges, `rst_n` is driven high, at which point the LFSR begins updating on each rising edge of `clk`. The stimulus sequence first holds `i_ptb_valid = 0` to observe the pure LFSR sequence, then periodically asserts `i_ptb_valid` while toggling `i_ptb`, and finally runs a short stress phase where `i_ptb` is randomized with `i_ptb_valid` held high. Waveforms are captured using `$dumpfile` and `$dumpvars` and viewed in a standard waveform viewer.

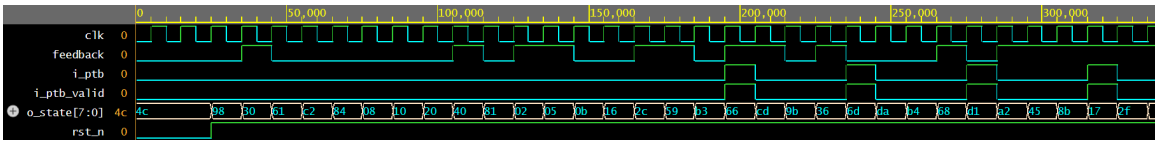


Figure 6.18: *`lfsr8.v` testbench waveform showing reset, pure LFSR operation, and `i_ptb` mixing over an extended run.*

The waveform in Figure 6.18 shows that, while `rst_n` is low, the output state remains at the reset value. On the first rising edge after reset deassertion, `o_state[7:0]` cleanly initializes to the non-zero seed `8'h4C` (`0100_1100`). No unknown or metastable values are observed. With `i_ptb_valid = 0`, the register then advances through a sequence of values such as

$$\texttt{4C} \rightarrow \texttt{98} \rightarrow \texttt{30} \rightarrow \texttt{61} \rightarrow \texttt{C2} \rightarrow \texttt{84} \rightarrow \texttt{08} \rightarrow \texttt{10} \rightarrow \texttt{20} \rightarrow \texttt{40} \rightarrow \texttt{81} \rightarrow \ldots$$

Each transition corresponds to a left shift of the previous state with the least-significant bit replaced by the feedback value.

The feedback network uses taps `o_state[7]`, `o_state[5]`, `o_state[4]`, and `o_state[3]` and computes the XNOR of these bits, optionally flipped by the term `i_ptb &`

i_ptb_valid. To confirm this behavior, tap values and feedback were extracted from the waveform at several points and compared against the RTL expression. An example is shown below for the pure LFSR phase where i_ptb_valid = 0:

Current o_state	Tap Bits [7,5,4,3]	XNOR Feedback	Next o_state
8'h4C (0100 1100)	0,0,0,1	0	8'h98 (1001 1000)
8'h30 (0011 0000)	0,1,1,0	1	8'h61 (0110 0001)

Figure 6.19: *Example LFSR phase when i_ptb_valid = 0*

In both cases, the next state observed in the waveform matches the theoretical result of shifting left and inserting the computed feedback bit, confirming that the XNOR tap logic is implemented correctly.

Later in the trace, i_ptb_valid pulses periodically and then remains high while i_ptb toggles or is randomized. The feedback signal and o_state sequence visibly diverge from the baseline path only on cycles where i_ptb_valid is asserted, demonstrating that the external bit is mixed in exactly when enabled and has no effect otherwise. Across the entire simulation window, o_state never enters the illegal all-zero state, reset behavior is deterministic, and the state progression remains well-formed under all stimulus patterns. These observations provide strong evidence that lfsr8.v meets its intended functional specification.

6.7.2.5 ring_oscillator.v Verification Process

The ring_oscillator module was mainly verified through behavioral observation rather than stimulus-driven testing, since this module is self-oscillating and does not depend on a testbench clock source. It retains a value and inverts it, which results in no output. The high switching in the ring oscillator acts as a noise generator, and this results in hiding the power traces from any attacker. To test this module, it additionally requires a method that can measure the power traces to find differences between the pre-modified core and the post-modified core. This is further explained in Chapter 7: System Testing, where the power traces can accurately be measured through simulation rather than through functional verification of the module.

Although there is limited visibility through functional verification with a testbench for this module, UVM verification still plays an important role. It ensures that the module works as intended through code coverage, can react properly to enable signals, and can be integrated into the broader system. The functional verification of the oscillation behavior is deferred to the post-synthesis simulation, switching activity analysis, and power-trace comparison rather than relying solely on traditional RTL-level checks.

6.8 Ibex UVM Testbench Architecture

In addition to the RTL design of the core itself, our project relies heavily on Ibex's UVM-based verification toolchain. The DV environment is organized as a conventional UVM testbench: a top-level module instantiates the Devices Under Test (DUT), connects to verification interfaces, and invokes a run_test function to launch UVM tests. At the

UVM level, an environment class manages multiple agents, scoreboards, and co-simulation components.

The DUT in our case is a specific configuration of the `ibex_top` core, extended with our project-specific modifications. This core is connected to UVM interfaces representing instruction and data memory ports, interrupt lines, and debug/reset control signals. The environment is designed around the idea of running program-driven tests: each test will load a binary into the testbench memory model, allow the core to execute it, and then monitor all external interface activity and architectural states for correctness.

6.8.0.1 Memory Interface Agents and Shared Memory Model

Ibex uses separate interfaces for instruction fetches and data load/store operations. The DV environment captures each of these interfaces within a dedicated memory agent. Each agent includes a driver, which responds to requests from the core with configurable response timing and optionally injected error responses, and a monitor, which observes all transactions and publishes them to analysis ports for scoreboards and coverage analysis. Both agents leverage a shared memory model instance that represents main memory from the perspective of the testbench. Upon initialization of each test, the program binary produced within the toolchain (typically through RISC-V-DV) is loaded into this main memory. The instruction and data agents then use the same backend, providing a coherent view of memory to the core and simplifying checks on the behavior of the included memory systems.

Because these memory agents and models are implemented in SystemVerilog and UVM, they are considered part of the software design of the verification environment, as opposed to the hardware design of the core, which is also implemented in SystemVerilog.

6.8.0.2 Interrupt and Debug Stimulus

In digital design, asynchronous events and logic are a common source of bugs, especially within larger designs such as an SoC. To address this, the DV environment includes support for interrupt and debug stimulus. An interrupt agent drives the core's interrupt lines with randomized and/or constrained patterns, allowing tests to target scenarios such as back-to-back interrupts, overlapping interrupts of different types, and interrupts that arrive prior to instruction execution completion (for multi-cycle instructions). In addition, debug and reset signals are typically controlled directly by sequences through virtual interfaces, enabling tests where the core can enter and exit debug mode at specific pipeline points, as well as tests in which reset is asserted asynchronously in the middle of instruction execution.

Once again, these components are considered testbench software elements that we are able to configure and extend for our needs, particularly when exercising our custom side-channel-oriented extensions under interrupt pressure.

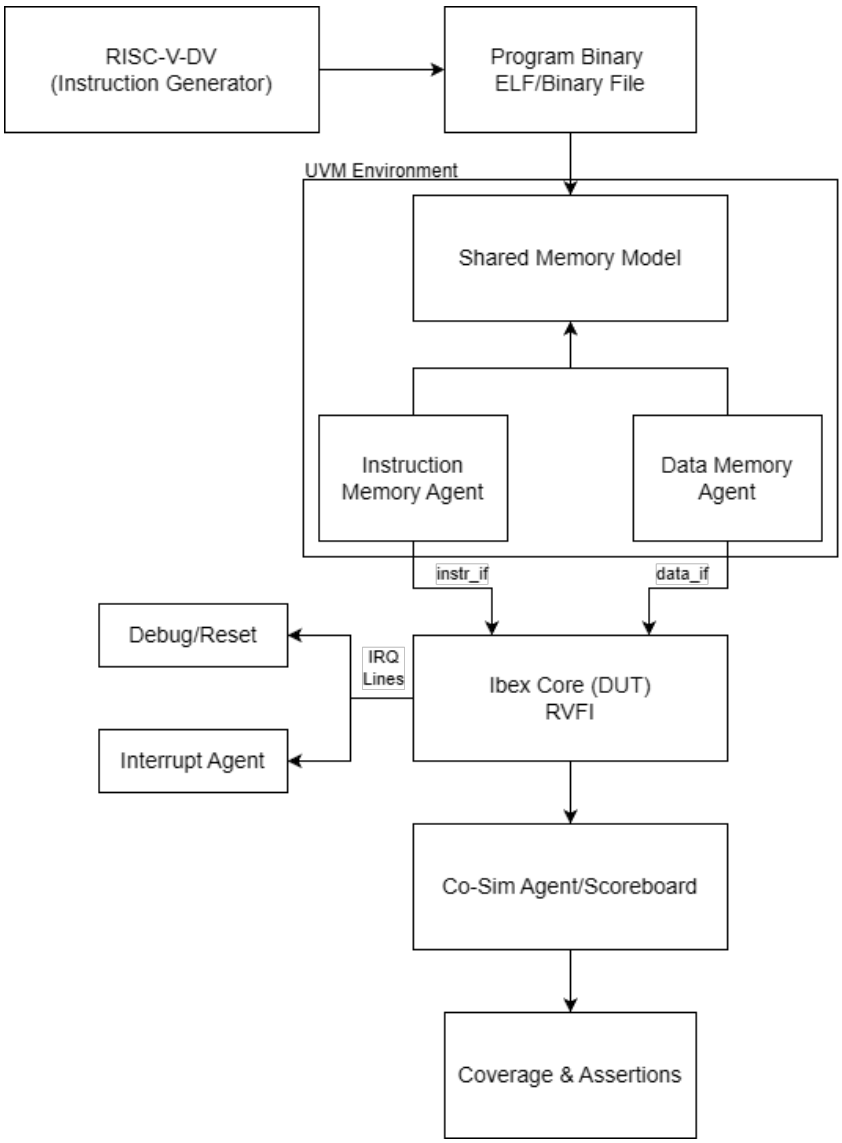


Figure 6.20: *Ibex UVM Testbench Architecture Overview*

6.8.0.3 Integration with RISC-V-DV

In order to generate unique, rich instruction streams, the DV environment incorporates RISC-V-DV, an open-source random instruction generator for the RISC-V ISA. In a typical test flow, RISC-V-DV first generates an assembly program subject to user-specified constraints (for example, instruction mix, CSR usage, and privileged operations). This assembly program is then compiled into a binary file using the RISC-V GCC toolchain. The UVM test loads the resulting binary into a shared memory model and configures the core's initial register and CSR states before executing the simulation, during which monitors and the co-simulation system continuously check the core's behavior.

From a software design perspective, RISC-V-DV can be considered an external tool that is integrated into our simulation flow through scripts and test classes: UVM tests select and parameterize the RISC-V-DV generators, and the resulting binaries become input

artifacts to the testbench.

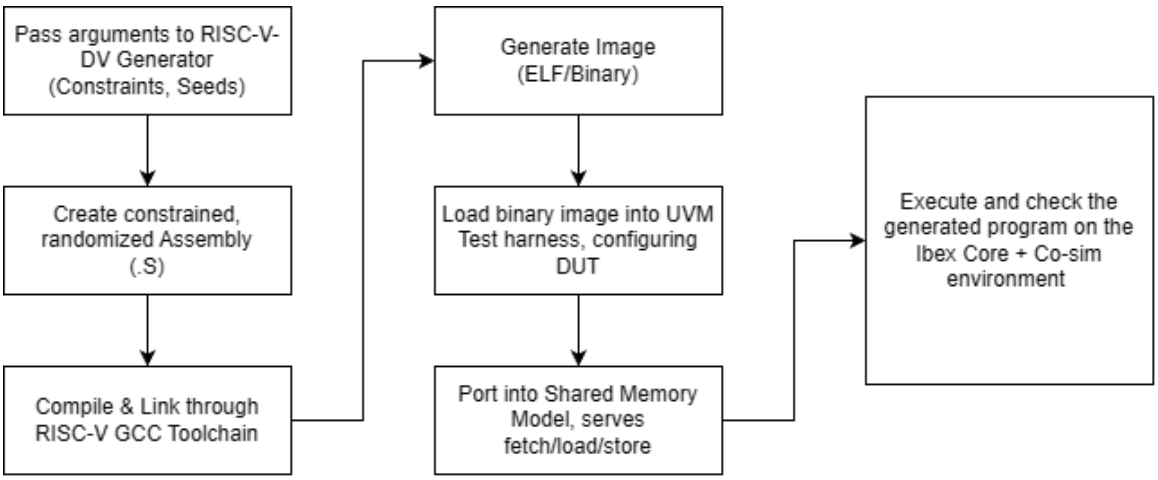


Figure 6.21: *RISC-V-DV Test Generation and Integration Flow*

In addition to verifying the functional correctness of interrupt handling, the DV environment was also used to measure average interrupt latency. UVM monitors were instrumented to record the number of clock cycles elapsed between assertion of an interrupt line and the retirement of the first instruction of the corresponding interrupt service routine, as identified via the RVFI interface. Across a range of randomized interrupt stimulus scenarios, these measurements were collected and averaged to produce an empirical interrupt latency figure, which was then compared against the cycle requirement specified in §2.3.

6.8.0.4 Co-Simulation Architecture

The co-simulation system is implemented as a combination of C++ and SystemVerilog UVM components, and is therefore largely part of the project's software components. At its core is a C++ co-simulation library that wraps a lowRISC-maintained fork of the Spike RISC-V ISS. This library provides an API to initialize a Spike instance with a given program image and initial system state, step the ISS by one instruction at a time, and query the ISS's register file, CSRs, and memory behavior. A minimal Direct Programming Interface (DPI) layer exposes these operations as SystemVerilog-callable functions. From the UVM environment's perspective, co-simulation is accessed through this DPI handle: testbench classes call functions such as "initialize co-sim," "step one instruction," or "check state," passing in the RTL-observed events for comparison.

On the SystemVerilog side of things, a co-simulation agent and associated scoreboard subscribe to RISC-V Formal Interface (RVFI) transactions, which describe each retired instruction (including PC, opcode, operands, and flags), as well as to memory transactions observed by the instruction and data memory monitors. For each retired instruction, the scoreboard triggers a co-sim step within Spike and checks that the ISS retired the same instruction at the same PC, that register writebacks, CSR updates, and interrupt behavior match, and that memory addresses and access types observed within RTL agree with Spike's view of the program.

We treat this co-simulation stack as part of the overall software design, as it dictates how our tests are implemented and how results are checked. Any extensions introduced may

require corresponding updates either to the co-sim library or to the checking logic in the scoreboard.

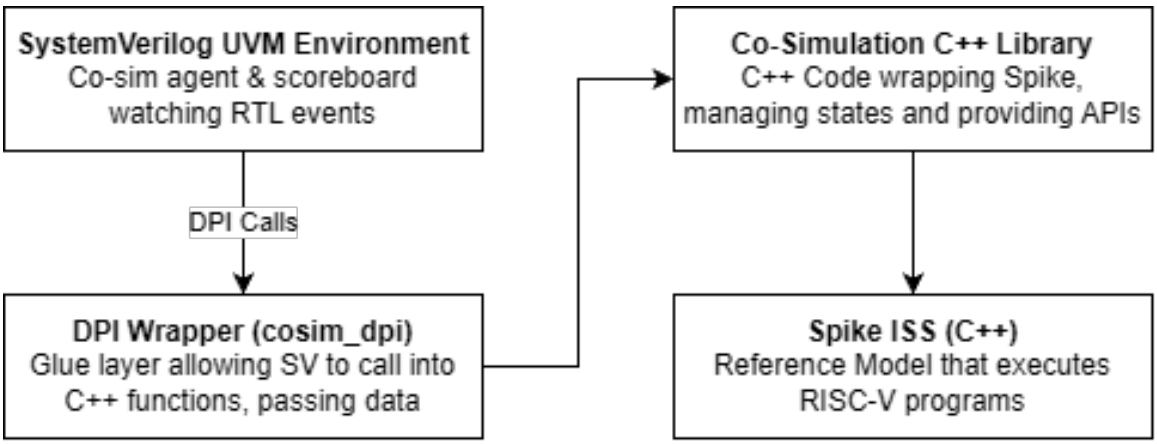


Figure 6.22: *Co-Simulation Software Stack for Ibex and Spike*

While Figure 6.22 shows the static software stack used for co-simulation, it is also helpful to view how these components interact dynamically. Figure 6.23 illustrates the per-instruction sequence of events during co-simulation, from an instruction retiring in Ibex through the RVFI monitor, DPI bridge, and Spike ISS, and back to the UVM scoreboard for comparison.

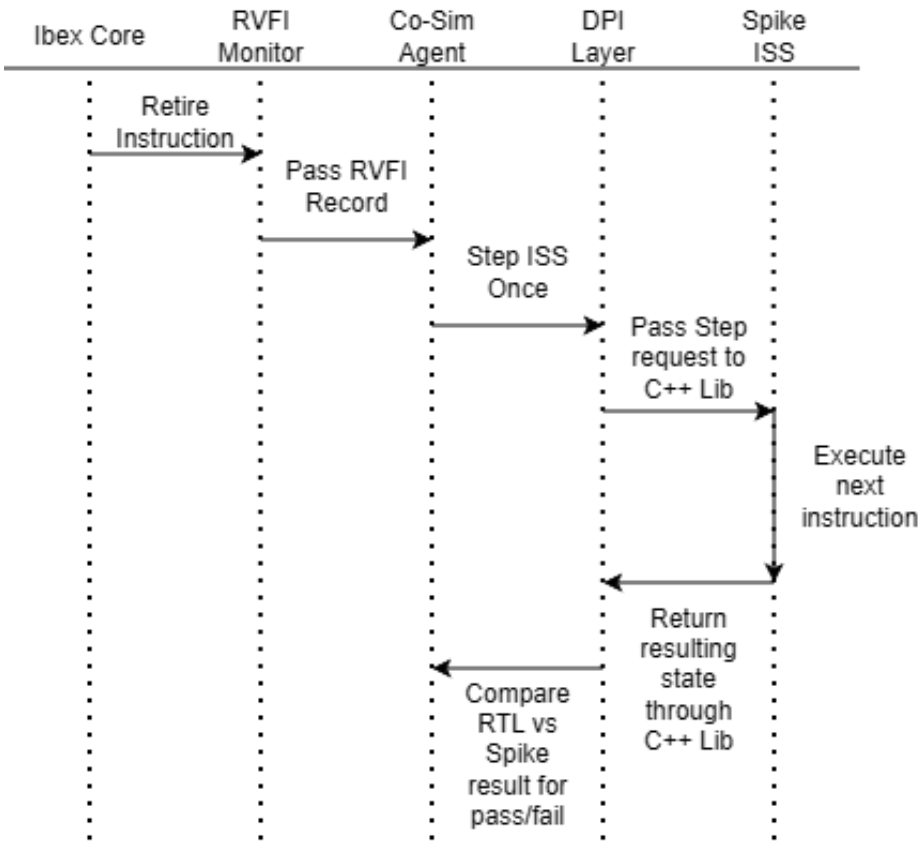


Figure 6.23: *Per-Instruction Co-Simulation Sequence Between Ibex and Spike*

6.9 Realization of the DV and Co-Simulation Environment

Regarding DV and co-simulation, much of what has been described in this chapter focuses on the theoretical capabilities of the infrastructure that is pre-packaged with the Ibex core. On paper, this is highly beneficial to the direction of our project, but these capabilities must be something that we can actually implement and extend in practice. In our case, this means setting up the full Ibex DV and co-simulation environment, together with all required dependencies and workflows, on the UCF VLSI servers.

At a high level, the workflow for realizing the DV + co-simulation environment is as follows. First, we must ensure that a sufficiently recent Python version is available. We then clone the upstream Ibex repository, install a compatible RISC-V GCC toolchain, build the lowRISC Spike fork used for co-simulation, configure and build the DV environment, and finally assign all expected environment variables and paths. Because environment variables are not retained between login sessions on the shared servers, we ultimately encapsulate these assignments in a setup script.

```
[al663069@vlsi ~] python3 --version
Python 3.6.8
[al663069@vlsi ~] source /home/net/al663069/ibex/ibex-venv/bin/activate
(ibex-venv) [al663069@vlsi ~] python3 --version
Python 3.10.14
```

Figure 6.24: *Verification of the local Python virtual environment on* ***vlsi1****.*

Although these steps are conceptually straightforward, they are complicated by the fact that we do not have administrative privileges on the UCF VLSI servers. The most Cadence-stable machine, `vlsi1`, provides Python 3.6.8 by default, whereas the Ibex DV infrastructure requires at least Python 3.10. On a machine where we control the operating system, this would typically be resolved by running commands such as `sudo apt update` and `sudo apt upgrade` to update the system packages. On a shared academic platform where we are not administrators, however, such approaches are not appropriate.

To resolve this, we opted to use Python virtual environments (`venv`) and a local, user-space Python installation. We created a dedicated directory in our home area, downloaded a suitable Python 3 source tarball via `wget`, unpacked it, and configured it to install into our home directory. After building with `make` and `make install`, we obtained a local Python instance that was not added to the global system path. Using this local Python binary, we then created a virtual environment via `python3 -m venv <env_dir>`, which produced an `activate` script. By sourcing this script, we enter a virtual environment in which the new Python version is used and all Python packages are installed locally, without affecting other users.

Once the Python environment was in place, the remaining setup largely followed the upstream documentation. We cloned the original Ibex repository using `git` and, within this local clone, installed the required Python dependencies via `pip install -U -r python-requirements.txt`.

The next major step was installing the RISC-V GCC toolchain. In a typical Linux installation with administrative access, one might simply run a command such as `sudo apt install gcc-riscv64-unknown-elf` (or a similar package). On the UCF VLSI servers, we again lack `sudo` privileges, and we also have an additional constraint: we specifically need a RISC-V GCC toolchain that targets the RV32IMC instruction set used by Ibex. The general RISC-V GCC ecosystem contains many different builds with varying ISA support, so we needed to identify an appropriate pre-built toolchain.

After some investigation, we selected the lowRISC toolchain release `20220210-1`, which provides an RV32IMC-focused GCC toolchain suitable for Ibex [76]. We downloaded the archive using `wget`, unpacked it in our home directory, and obtained the necessary binaries, such as `riscv32-unknown-elf-gcc` and `riscv32-unknown-elf-objcopy`.

With the toolchain installed, we configured the environment variables expected by the Ibex DV flow. Specifically, we set `RISCV_TOOLCHAIN` to point to the toolchain root directory, `RISCV_GCC` to the `riscv32-unknown-elf-gcc` binary, and `RISCV_OBJCOPY` to the `riscv32-unknown-elf-objcopy` binary. We also added the toolchain `bin/` directory to our `PATH`. These assignments were later collected into a shell script so that the environment can be re-established quickly in new login sessions.

The final major step was building the lowRISC Spike fork used by the co-simulation infrastructure. This process was relatively straightforward: we cloned the corresponding repository, checked out the required revision, configured the build, and then invoked `make` followed by `make install`. To integrate Spike with the Ibex DV environment, we set

two additional environment variables: `SPIKE_PATH`, which points to the installed Spike binary, and `PKG_CONFIG_PATH`, which allows the build system to locate the Spike libraries via `pkg-config`. These variables were also added to our setup script.

With Python, the RV32IMC GCC toolchain, and the Spike fork configured in this way, we obtained a reproducible DV and co-simulation environment on the UCF VLSI servers that can be reinitialized quickly and extended to support our modified core.

It should be noted that while the DV and co-simulation environment served as the primary platform for pre-synthesis functional verification and for measuring average interrupt latency, post-synthesis gate-level simulation was ultimately conducted using the Ibex Simple System instead. This decision was driven by the nature of our Boolean masking implementation, which involves post-synthesis modifications to the netlist via SERV-style Tcl edits; more specifically, these are changes that are not reflected in the RTL and therefore cannot be meaningfully simulated within the RTL-centric DV environment.

6.9.1 Using the Ibex DV and Co-Simulation Environment in Practice

With the Ibex DV and co-simulation environment successfully installed on the UCF VLSI servers, the next step is to exercise it as intended: running tests, generating waveforms, and interpreting log output. This section summarizes how we interact with the environment in day-to-day use, starting from a single command-line invocation and ending with a repeatable debug workflow that we will later apply to our modified core.

6.9.1.1 Running a Single UVM Test

In our setup, tests are launched from the `dv/uvm/core_ibex` directory using a Make-based flow. A typical command to run a single test under Xcelium is:

```
make SIMULATOR=xlm \
     TEST=riscv_machine_mode_rand_test \
     ITERATIONS=1 \
     COSIM=1 \
     WAVES=1 \
     OUT=out_xlm_cosim_gui_demo
```

Here, `SIMULATOR=xlm` selects the Xcelium (XLM) backend provided by the UCF VLSI servers, and `TEST=` chooses a specific UVM test from the Ibex testlist. The `ITERATIONS` parameter controls how many independent random seeds are run, while `COSIM=1` enables lockstep co-simulation with the Spike ISS. Setting `WAVES=1` instructs the flow to generate a waveform database suitable for SimVision, and `OUT=` selects the output directory where logs and artifacts are written.

6.9.1.2 Co-Simulation and Waveform Generation

When `COSIM=1` is enabled, each retired instruction observed via the RVFI interface is forwarded to the co-simulation bridge, which steps the Spike ISS and compares its architectural state against the RTL. Any divergence is reported immediately in the UVM log and summarized in the regression report.

Waveforms are generated whenever `WAVES=1` is passed to the Make command. For the Xcelium backend, this produces a SimVision-compatible database (`waves.shm`) in the selected `OUT` directory. We can then launch SimVision either directly from the `make` flow (for GUI runs) or by invoking `simvision` on the generated database. This allows us to inspect pipeline signals, register file updates, RVFI traces, and memory interface activity around any event of interest.

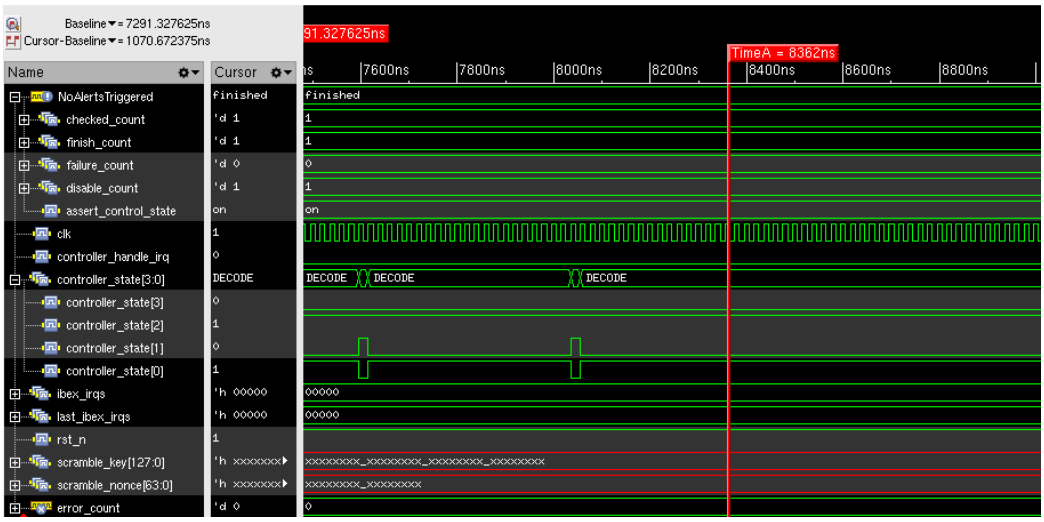


Figure 6.25: *Sample simvision waveform of* ***`riscv_machine_mode_rand_test`***

6.9.1.3 Interpreting Log and Regression Outputs

Each test run produces several log files. The UVM transcript records the sequence of phases, driver and monitor activity, and any assertion or protocol violations. A regression log (`regr.log`) summarizes, for each test and iteration, whether the run passed, failed due to a co-simulation mismatch, or terminated for another reason (for example, a timeout or fatal assertion).

```
100.00% PASS 1 PASSED, 0 FAILED
riscv_machine_mode_rand_test.26203: PASS


##################################################
# Details of failing tests
##################################################
No failing tests. Nice job!

##################################################
# Details of passing tests
##################################################

riscv_machine_mode_rand_test.26203
----------------------------------
binary:          test.bin
rtl_log:         rtl_sim.log
rtl_trace:       trace_core_00000000.log
iss_cosim_trace: spike_cosim_trace_core_00000000.log

[PASSED]
```

Figure 6.26: *Sample* `regr.log` *output of* `riscv_machine_mode_rand_test`

Alongside the typical regression files (`regr.log`), various other log and trace files exist (`trace_core.log` and `spike_cosim_trace_core.log`) which can be analyzed for any form of logical mismatch. Figures 6.27 and 6.28 below both visibly reflect similar memory addresses and instruction IDs executed from the `riscv_machine_mode_rand_test` sample test packaged within the DV + Co-sim environment.

```
80000080  40001c37  lui x24,0x40001  x24=0x40001000
80000084  106c0c13  addi  x24,x24,262  x24:0x40001000 x24=0x40001106
80000088  301c1073  csrrw x0,misa,x24  x24:0x40001106  x0=0x00000000
8000008c  00026d17  auipc x26,0x26   x26=0x8002608c
80000090  c58d0d13  addi  x26,x26,-936   x26:0x8002608c x26=0x80025ce4
80000094  00002c17  auipc x24,0x2  x24=0x80002094
80000098  46cc0c13  addi  x24,x24,1132   x24:0x80002094 x24=0x80002500
8000009c  001c6c13  ori x24,x24,1  x24:0x80002500 x24=0x80002501
800000a0  305c1073  csrrw x0,mtvec,x24   x24:0x80002501  x0=0x00000000
```

Figure 6.27: *sample* `trace_core.log` *output of* `riscv_machine_mode_rand_test`

```
1 core   0: 0x80000080 (0x40001c37) lui     s8, 0x40001
2 core   0: 3 0x80000080 (0x40001c37) x24 0x40001000
3 core   0: 0x80000084 (0x106c0c13) addi    s8, s8, 262
4 core   0: 3 0x80000084 (0x106c0c13) x24 0x40001106
5 core   0: 0x80000088 (0x301c1073) csrw    misa, s8
6 core   0: 3 0x80000088 (0x301c1073) c769_misa 0x40901104
7 core   0: 0x8000008c (0x00026d17) auipc   s10, 0x26
8 core   0: 3 0x8000008c (0x00026d17) x26 0x8002608c
9 core   0: 0x80000090 (0xc58d0d13) addi    s10, s10, -936
0 core   0: 3 0x80000090 (0xc58d0d13) x26 0x80025ce4
1 core   0: 0x80000094 (0x00002c17) auipc   s8, 0x2
2 core   0: 3 0x80000094 (0x00002c17) x24 0x80002094
3 core   0: 0x80000098 (0x46cc0c13) addi    s8, s8, 1132
4 core   0: 3 0x80000098 (0x46cc0c13) x24 0x80002500
5 core   0: 0x8000009c (0x001c6c13) ori     s8, s8, 1
6 core   0: 3 0x8000009c (0x001c6c13) x24 0x80002501
7 core   0: 0x800000a0 (0x305c1073) csrw    mtvec, s8
8 core   0: 3 0x800000a0 (0x305c1073) c773_mtvec 0x80002501
```

Figure 6.28: *sample* `spike_cosim_trace_core.log` *output of* `riscv_machine_mode_rand_test`

In the co-simulation case, failures are typically annotated with the program counter and instruction opcode at the point of divergence, together with details such as mismatched register write-back values or differing CSR contents. These log entries provide a compact starting point for targeted debugging in the waveform viewer.

6.9.1.4 Typical Debug Workflow

Our typical debug workflow begins by scanning `regr.log` to identify any failing tests and the associated random seed. We then open the corresponding UVM log to locate the first reported error, which usually includes the failing instruction and its program counter. Using this information as a time reference, we load the appropriate waveform database into SimVision and navigate to the failure point.

From there, we correlate the RVFI trace, register file write-backs, and memory transactions against the Spike co-simulation checks, verifying whether the discrepancy is due to a misunderstanding of the test, a configuration issue, or a genuine RTL bug. This is the same workflow we intend to follow when extending the environment with modified-core-specific instructions and side-channel-oriented features.

6.9.1.5 Adapting Tests for Modified Core

Because this build and debug workflow is already in place, integrating pre-synthesis functional tests for these extensions becomes an exercise in adapting test definitions and scoreboard logic. Post-synthesis verification, however, is handled separately using the Ibex Simple System environment, as described in §6.10, due to the netlist-level nature of the Boolean masking implementation.

6.10 Post-Synthesis Verification

6.10.1 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 [39]. 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, because our Boolean masking is inserted post-synthesis via Tcl edits to the synthesized netlist, the RTL-centric DV and co-simulation environment is not suitable for this verification step. Instead, gate-level simulation is conducted using the Ibex Simple System, which provides a lightweight SoC context that can be compiled and simulated directly from the post-synthesis netlist without relying on the UVM infrastructure.

Given the project scope, the verification effort primarily focused on two practical checks. First, we used logical equivalence checking (LEC) [77], where applicable, to demonstrate that the post-synthesis and post-masking netlists preserved the intended architectural and logical behavior. Second, we ran zero-delay gate-level simulations through the Ibex Simple System to confirm that the modified design could still execute realistic program scenarios correctly. These simulations provided confidence that reset behavior, CSR/PMP interactions, and instruction execution remained functionally valid after synthesis and masking. While SDF back-annotated timing simulation and detailed latency-distribution analysis remain valuable future verification steps, they were not the primary focus of the completed verification workflow.

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.

6.10.2 Execution Stimulus Model (C to VMEM Flow)

In this verification flow, software stimulus is generated by compiling bare-metal C tests into RISC-V binaries and converting them into VMEM images, which are loaded into Simple System memory at simulation start. Gate-level simulation is then executed by running these VMEM-backed programs on the netlist-integrated Simple System platform. As a result, functional coverage reflects behaviors reachable through software-driven execution

over the exposed memory map and peripherals, rather than constrained-random sequence generation. This execution model is important context for coverage interpretation, including unreachability classification, because any behavior requiring unavailable interfaces or non-instantiated platform features cannot be stimulated in this environment.

6.10.3 Functional Coverage and Unreachability Analysis

To systematically verify behavior in gate-level simulation, we implement a functional coverage model directly in the Simple System environment using SystemVerilog covergroups bound into the core wrapper. Because this environment is not UVM-based, coverage is collected from in-testbench and bound covergroups rather than from UVM subscriber components.

The current model is organized around what the present GLS flow can directly observe. Stage-oriented covergroups track IF, ID, EX, and WB activity, instruction completion, branch decisions, LSU busy behavior, and opcode-class execution. Interface-oriented covergroups capture instruction and data channel handshake behavior, request/response activity, and read-versus-write outcomes. Addressing and MMIO covergroups classify access region (RAM, simulator control, timer, other), byte-enable patterns, alignment class, and MMIO register targeting, including read/write crosses.

Coverage data from individual GLS tests is then processed in Cadence IMC, which is the tool used to load runs, inspect detailed covergroup/bin results, and merge multiple test databases into a combined coverage view for regression-level assessment. IMC-generated reports are the primary basis for coverage closure tracking in this flow.

Uncovered-bin disposition is also handled in IMC. Unreachability analysis is performed against the actual stimulus capabilities of the Simple System platform, and bins deemed structurally impossible for this environment are marked as unreachable in the coverage dataset through IMC exclusion and unreachability mechanisms. This separates true stimulus gaps from platform-impossible bins, so closure is measured against the achievable coverage space rather than an idealized model that assumes unavailable stimulus sources.

6.11 Physical Design

The work completed prior to physical design has primarily emphasized front-end VLSI development, including the specification and modification of the Ibex-based RTL to create the new modified core, along with verification through design validation (DV) and co-simulation environments. These efforts have established a functionally correct and validated design baseline.

Building on this foundation, the focus now shifts toward physical design and implementation. The objective of this phase is to systematically translate the verified RTL into a manufacturable layout by progressing through the complete digital implementation flow, from synthesis through physical design and final sign-off, ultimately generating a GDSII file suitable for fabrication.

The descriptions in this section are organized by topic and follow the "Digital Realization of the RTL-to-GDSII" flow presented in Cadence's RTL-to-GDSII v7.0 training module [78].

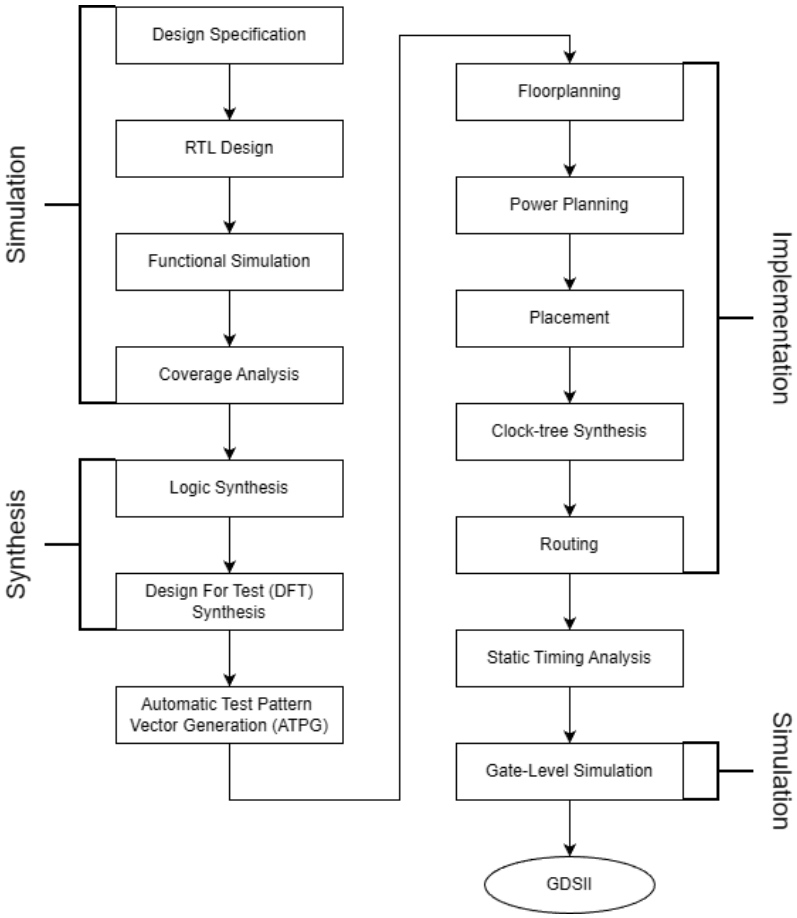


Figure 6.29: *Realization of RTL-to-GDSII Flow. Outlines the approximate, generalized workflow from start-to-finish for realizing a GDSII file output.*

6.11.1 Simulation: From Design Specification to Gate-Level Behavior

The starting point for Senior Design II was a stable design specification and RTL for the modified Ibex core. By the end of Senior Design I, we fully developed the microarchitectural changes, and interface behavior captured in a configuration-controlled Ibex RTL tree that we treat as "frozen" for physical implementation. Around this RTL, we continued to rely on functional simulation using the existing Ibex DV environment and Spike-based co-simulation for pre-synthesis RTL validation.

Within this simulation-focused portion of the flow, we will pay attention not just to pass/fail behavior, but also to coverage. Code coverage (line, toggle, branch) and functional coverage (e.g., instruction mix,modified-core corner cases) will be used to quantify how thoroughly the modified core logic has been exercised. As we move into later stages of the flow, we will also extend our simulation to gate-level models: after synthesis and, later, after layout (for gate-level simulation) we will use the Ibex Simple System environment, for post-synthesis verification of the Boolean masking implementation. These gate-level simulations create a bridge between purely functional RTL verification and the timing-/parasitic-aware behavior of the implemented design.

6.11.2 Synthesis: Logic and Design-for-Test (DFT)

Once the RTL and simulation environment are mature, the next topic is synthesis. Logic synthesis will map the modified Ibex RTL onto a target standard-cell library using constraints expressed in SDC (clock definitions, I/O constraints, and design-specific exceptions). This step produces a gate-level netlist and preliminary quality-of-results (QoR) metrics such as cell area, estimated critical-path delay, and early power estimates. Where possible, we will use logic equivalence checking (LEC) between the RTL and the synthesized netlist to confirm that synthesis has not altered the intended behavior.

In a standard industrial flow, logic synthesis is followed by design-for-test (DFT) synthesis, where scan chains and other test structures are inserted into the netlist to enable high fault coverage during manufacturing test. For our project, we plan to at least document the role of DFT in the digital realization flow and, time and tool access permitting, perform a small-scale DFT experiment (for example, scan insertion on a cut-down modified core) to demonstrate how the design would be prepared for production test without fundamentally changing the functional behavior observed at the RTL level.

6.11.3 Automatic Test Pattern Generation (ATPG)

Following DFT synthesis, the DFT-augmented netlist is used as the basis for automatic test pattern generation (ATPG). ATPG tools analyze the structural netlist and inserted scan architecture to create sets of test vectors capable of detecting a wide range of manufacturing defects (e.g., stuck-at or transition faults) with high coverage. While full ATPG sign-off may be beyond the scope of Senior Design II, we will treat ATPG as an important conceptual step in the digital realization flow and, where possible, generate example patterns for the modified core. This highlights how the design moves from being purely functionally correct to being testable and screenable in silicon.

6.11.4 Physical Implementation

The next major topic is physical implementation, in which the synthesized (and possibly DFT-augmented) gate-level netlist is realized as a concrete physical layout. We will begin with floorplanning, choosing a core area, aspect ratio, and high-level placement of any macros such as instruction and data memories. At this stage we also introduce global I/O placement and consider how the modified core might sit within a larger SoC context. Power planning overlays a power and ground grid on this floorplan, ensuring adequate IR-drop performance and providing a solid backbone for clock distribution.

6.11.4.1 Floorplanning

Floorplanning establishes the physical foundation of the design by defining the die size, core area, utilization, and aspect ratio. These parameters are selected to balance area efficiency with routability, ensuring that sufficient whitespace is available to alleviate congestion during placement and routing. A higher utilization improves area efficiency but increases the risk of routing congestion and timing degradation, while a lower utilization provides more flexibility at the cost of increased silicon area. Therefore, careful selection of core density is essential to achieve an optimal tradeoff between performance and manufacturability.

In addition to defining the core dimensions, floorplanning includes the placement of large macros such as instruction and data memories. These macros are positioned to minimize critical path distances and to reduce routing complexity between frequently communicating blocks. Consideration is also given to alignment with standard-cell rows, routing layer access, and future clock and power distribution. A well-constructed floorplan provides a structured layout that guides all subsequent physical design stages and directly impacts timing, congestion, and overall design quality.

6.11.4.2 Pin Placement

Pin placement determines the physical locations of input and output ports along the boundaries of the design. This step is critical in shaping the overall routing complexity, as inefficient pin placement can result in long interconnect paths, increased congestion, and degraded timing performance. Pins are typically distributed along the periphery in a manner that aligns with the logical data flow of the design, minimizing cross-chip signal traversal and reducing wirelength.

In this design, pin placement is approached with both internal and system-level considerations in mind. Internally, pins are positioned to reduce the distance between related functional blocks, improving timing and easing routing constraints. Externally, pin locations are selected to facilitate integration with higher-level modules in a potential SoC environment. By carefully organizing pin locations, the design achieves improved routability, reduced congestion, and more predictable timing behavior.

6.11.4.3 Power Planning

Power planning focuses on the construction of a robust power delivery network (PDN) capable of supplying stable voltage across the entire chip. This involves creating power rings around the core and inserting horizontal and vertical power stripes across multiple metal layers. These structures distribute power evenly throughout the design and provide low-resistance paths for current flow, which is essential for maintaining voltage stability during switching activity.

A key objective of the PDN design is to minimize IR drop and ensure reliable operation under worst-case conditions. This requires careful selection of stripe widths, spacing, and metal layer usage, as well as consideration of current density and electromigration limits. Additionally, the PDN must be designed to coexist efficiently with signal routing resources, avoiding excessive blockage of routing tracks. A well-designed power network not only ensures functional correctness but also supports timing closure and overall design robustness.

6.11.4.4 CTS and Routing

Clock tree synthesis (CTS) is performed after placement to construct a balanced clock distribution network that meets skew and insertion delay constraints. The clock network is one of the most critical components of the design, as it synchronizes all sequential elements. During CTS, buffers and inverters are strategically inserted to ensure that the clock signal arrives at all endpoints within acceptable timing margins, minimizing skew and preventing timing violations.

ollowing CTS, routing is performed to connect all signal and clock nets while adhering to the design rules of the target fabrication process. This includes both global routing, which

determines approximate paths for interconnects, and detailed routing, which finalizes exact wire geometries. Routing must balance multiple objectives, including minimizing wirelength, reducing congestion, and mitigating signal integrity issues such as crosstalk and coupling noise. Iterative optimization is often required during this stage to resolve congestion hotspots and improve timing performance across the design.

6.11.4.5 Analysis & Signoff

The analysis and signoff stage verifies that the design satisfies all timing, physical, and functional requirements prior to fabrication. This includes design rule checking (DRC) to ensure compliance with manufacturing constraints and layout versus schematic (LVS) verification to confirm that the physical layout matches the intended circuit connectivity. These checks are essential for ensuring that the design is manufacturable and free of structural errors.

In addition to physical verification, static timing analysis (STA) is performed to validate that the design meets setup and hold timing constraints under worst-case operating conditions. Parasitic extraction is used to model the effects of interconnect resistance and capacitance, allowing for more accurate timing analysis. Power integrity checks, including IR drop and electromigration analysis, may also be conducted to ensure reliable operation. Together, these analyses provide confidence that the design is robust and ready for final GDSII generation and tape-out.

Our finalized physical design is shown below.

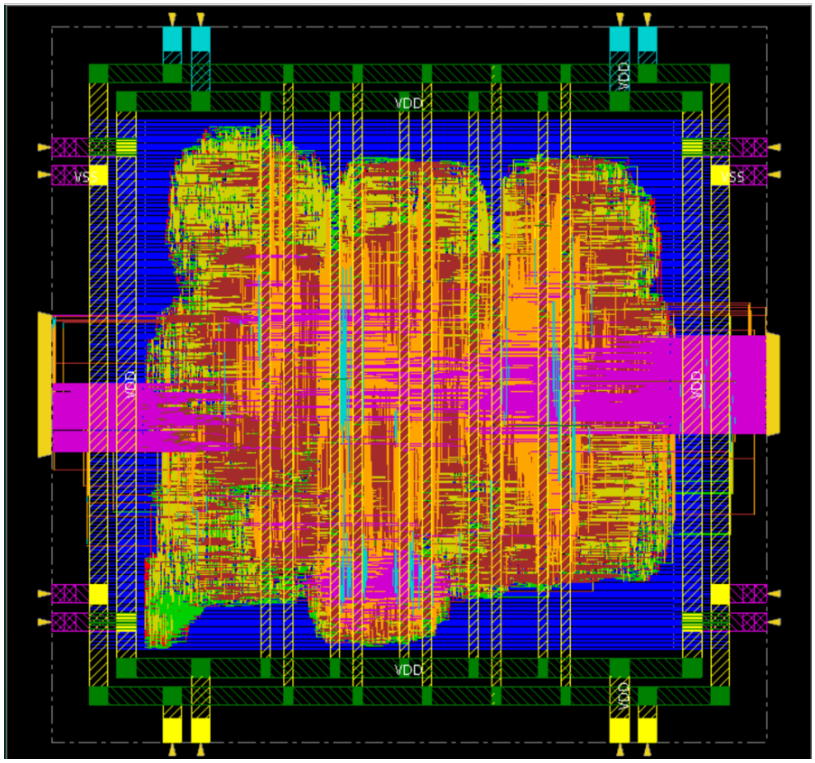


Figure 6.30: *Final Physical Design in Cadence Innovous.*

6.11.4.6 Post-Route Timing Analysis with Innovus

After place and route, timing analysis was performed on the design with Cadence Innovus to ensure that it still met timing requirements. This required checking for both setup and hold violations, where a setup violation means that signals are arriving too late and hold violations involve signals arriving too early. As shown in Figure 6.30, performing post-route timing analysis generates a report that shows timing violations on specific paths. Path 1 had a negative slack of -1.035 ns, which meant that the signal was arriving too late.

```
Path 1: VIOLATED (-1.035 ns) Setup Check with Pin u_ibex_core/u_masked/cs_registers_i/mi
nstret_counter_i/counter_q_reg[63]/cmo_flop1/CK->D
              View: slow_setup
             Group: CLK_GATED
       Startpoint: (R) randbits[0]
             Clock: (R) MAIN_CLK
         Endpoint: (R) u_ibex_core/u_masked/cs_registers_i/minstret_counter_i/counter_
q_reg[63]/cmo_flop1/D
             Clock: (R) CLK_GATED

                      Capture       Launch
       Clock Edge:+   100.000       50.000
       Src Latency:+   -0.865        0.000
       Net Latency:+    0.885 (P)    0.000 (I)
           Arrival:=  100.019       50.000

             Setup:-    0.108
     Required Time:=   99.911
      Launch Clock:=   50.000
       Input Delay:+    1.000
         Data Path:+   49.946
             Slack:=   -1.035
```

Figure 6.31: *Timing analysis results with negative slack violation of* $-1.035ns$

Resolving these timing errors involves performing Static Timing Analysis (STA) which will be covered in the following section. Through these methods, this negative slack as a result of setup time violation was eliminated ensuring our design was meeting timing. The final slack after these fixes is shown in the next section.

```
------------------------------------------------------------------
         time_design Summary
------------------------------------------------------------------

Setup views included:
 slow_setup

+--------------------+---------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg |reg2cgate| default |
+--------------------+---------+---------+---------+---------+
|           WNS (ns):|  0.000  | 42.842  | 41.367  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |    0    |
|          All Paths:|  7700   |  5115   |    1    |  6636   |
+--------------------+---------+---------+---------+---------+
```

Figure 6.32: *Final post-route timing results with* 0 *violations across all paths*

6.11.5 Static Timing Analysis (STA)

Static Timing Analysis (STA) is used to verify that all signal paths in a design meet timing requirements by checking for setup and hold time violations. Using *Cadence Innovus*, these violations can be efficiently addressed with the `opt_design` command, applying either the `-setup` or `-hold` option to target the corresponding constraint.

The optimization process resolves violations by automatically inserting buffers to improve signal propagation and resizing logic gates to satisfy timing constraints, ultimately achieving zero or positive slack across all paths.

After optimization, a timing report is generated to evaluate design performance. Figure 6.33 shows the Worst Negative Slack (WNS) of the design. As illustrated, no negative slack remains, indicating that all timing requirements have been successfully met.

```
Path 1: MET (0.000 ns) Latch Borrowed Time Check with Pin core_clock_gate_i/gen_generic.u_impl_generic/en_latch_reg/GN->D
              View: slow_setup
             Group: CLK_RAND
       Startpoint: (R) randbits[7]
             Clock: (R) MAIN_CLK
         Endpoint: (F) core_clock_gate_i/gen_generic.u_impl_generic/en_latch_reg/D
             Clock: (F) CLK_RAND

                     Capture       Launch
      Clock Edge:+    50.000       50.000
     Src Latency:+    -0.865        0.000
     Net Latency:+     0.343 (P)    0.000 (I)
         Arrival:=    49.478       50.000

    Time Borrowed:+     8.239
    Required Time:=    57.717
     Launch Clock:=    50.000
      Input Delay:+     1.000
        Data Path:+     6.717
            Slack:=     0.000
```

Figure 6.33: *Design meeting timing after STA*

Cadence Tempus is typically used as the primary timing sign-off tool in the final stages of the ASIC design flow. However, due to project time constraints, sign-off-level analysis with Tempus was not performed. Instead, Cadence Innovus was used for Static Timing Analysis (STA) throughout the implementation process.

In practice, STA drives many iterations during both synthesis and physical implementation. Paths that fail or are close to failing timing constraints often require design refinements, such as constraint adjustments, buffer insertion, gate resizing, or localized modifications to placement and routing. STA results, in conjunction with physical metrics such as congestion and IR drop, ultimately determine whether this project's core can reliably achieve its target operating frequency. Although final sign-off with Tempus remains future work, STA was performed after routing to ensure that timing requirements were consistently met.

6.11.6 Logic Equivalence Checking (LEC) Across Synthesis and Implementation

Logic equivalence checking (LEC) is used to prove that two representations of the design are functionally identical. In our flow, we primarily view the RTL as the golden model and treat the synthesized gate-level netlist as the revised model. After logic synthesis (and after any major DFT or ECO changes), we would use LEC to mathematically confirm equivalence between the RTL and the gate-level netlist that enters physical implementation.

Although physical implementation does not intentionally change logic functionality, many flows maintain both pre- and post-implementation netlists (for example, pre-CTS and post-route), and equivalence checking can be used between these netlists as well. In that sense, both the synthesis and implementation stages feed into LEC: synthesis produces the first gate-level representation to be proven equivalent to RTL, and implementation may introduce further optimized netlists whose correctness must still trace back to the original RTL. Together with STA and gate-level simulation, LEC forms the backbone of sign-off, ensuring that the final GDSII delivered at the end of the flow represents the same design that was verified at the RTL level.

Chapter 7 - System Testing and Evaluation

7.1 Use of Ibex DV and Co-Simulation for Pre-Synthesis System Testing

Our pre-synthesis system-level testing strategy is built primarily around the Ibex DV and co-simulation environment introduced earlier. For post-synthesis system testing (including validation of the Boolean masking implementation) we use the Ibex Simple System, which enables gate-level simulation directly on the modified netlist. At a high level, each system test begins with a test program that is generated (often via RISC-V-DV), compiled, and loaded into the UVM memory model. The core then executes this program under randomized memory timing, interrupt patterns, and debug events. Throughout execution, UVM monitors observe instruction retirements and external interface activity, while the co-simulation system steps the Spike ISS in lockstep and checks that the RTL and ISS behaviors match at each retired instruction. This process continues until a designated end condition is met, such as specific flags, CSR states, or memory signatures, or until a timeout is reached.

A test is considered passing only if no mismatches are reported by the co-simulation scoreboard, no protocol or assertion violations occur on the memory or interrupt interfaces, and the test's end condition is reached within the allowed simulation time. This approach allows us to treat the co-simulation system as a strong oracle for correctness, enabling us to detect subtle regressions introduced by our design modifications and extensions. It should be noted that results pertaining to gate-level behavior (including those related to the Boolean masking implementation) are evaluated using the Ibex Simple System environment rather than the DV/Co-sim infrastructure, as the masking modifications exist only at the post-synthesis netlist level.

7.1.1 Regression Strategy and Coverage

To evaluate our system thoroughly, we aim to run a regression suite that combines baseline Ibex tests with project-specific scenarios. The baseline set includes sanity checks, directed tests, and standard RISC-V-DV scenarios used to confirm that we have not broken existing behavior in the unmodified core. On top of this, we define project-specific tests that focus on our extensions, such as custom instructions, masking strategies, and noise injection mechanisms.

For each simulation run, we collect functional coverage information that characterizes

how the design was exercised, including instruction mix, CSR usage, flag and trap types, interrupt scenarios, and any known corner cases specific to our extensions. We also collect code coverage information, such as line, branch, and toggle coverage within the RTL, to gauge how thoroughly the core implementation has been exercised. System testing is therefore evaluated not only by the quantity of passing tests, but also by coverage metrics that reveal how much of the design space and microarchitectural behavior has been explored.

7.1.2 Evaluation of DV/Co-sim Results

This section presents the results of pre-synthesis functional verification conducted using the Ibex DV and co-simulation environment. Testing was structured around five RISC-V regression tests drawn from the riscv-dv suite, each executed across 8 unique seeds to ensure stimulus diversity. The modified Ibex core was verified against the baseline Ibex core using RVFI-based trace comparison, with SPIKE serving as the golden reference model for RISC-V compliance checking.

Across the regression suite, the modified core passed all seeds for four of the five test categories: `riscv_arithmetic_basic_test`, `riscv_rv32im_instr_test`, `riscv_illegal_instr_test`, and `riscv_machine_mode_rand_test` each produced 8 passing runs. The `riscv_rand_instr_test`, which exercises realistic workloads including context switching and CSR behavior, produced 4 passing and 4 mismatched runs. These mismatches were classified as false-negatives: both the baseline and modified cores diverge at the same `bge` branch instruction due to CSR-dependent, performance-sensitive timing differences, and all 4 mismatched seeds exhibit consistent divergence behavior, confirming that the mismatch reflects a known co-simulation artifact rather than a correctness failure in the modified design.

For interrupt latency measurement, the DV environment was extended with a custom testbench that periodically injects interrupts into riscv-dv generated programs, recording the cycle count from interrupt assertion to the first retired instruction of the corresponding handler via the RVFI interface. Five interrupt-focused tests were executed across 9 unique seeds each, with 200 interrupts injected per test/seed combination. Results are summarized in Table 7.1. The measured average interrupt latency across all tests was 15.1 clock cycles. The minimum observed latency of 6 cycles and maximum of 44 cycles were consistent across all five test configurations, suggesting that the latency distribution is dominated by pipeline drain and handler entry overhead rather than by test-specific stimulus patterns.

Table 7.1: *Interrupt Latency Measurement Results (DV/Co-sim, 9 Seeds, 200 Interrupts/Seed)*

Test Name	Description	Min	Max	Avg
`riscv_single_interrupt_test`	Issues interrupts one at a time, checks for proper return	6	43	15.2
`riscv_multiple_interrupt_test`	Issues multiple interrupts simultaneously, heavier stability test	6	43	16.0
`riscv_interrupt_wfi_test`	Issues interrupts following a WFI (wait-for-interrupt) instruction	6	44	10.7
`riscv_nested_interrupt_test`	Issues an interrupt while the interrupt handler is already executing	6	43	16.8
`riscv_interrupt_csr_test`	Issues interrupts during writes to interrupt-related CSRs	6	44	15.1

Taken together, the regression and latency results demonstrate that the modified core preserves functional correctness across the baseline RV32IMC instruction set under randomized stimulus, with the false-negative classification of the `riscv_rand_instr_test` mismatches supported by consistent cross-seed evidence. The interrupt latency result represents the one outstanding gap against the defined requirements, and is carried forward as a tracked item for SD2. It should be noted that results pertaining to gate-level behavior, including those related to the Boolean masking implementation, are evaluated using the Ibex Simple System environment rather than the DV/Co-sim infrastructure, as the masking modifications exist only at the post-synthesis netlist level.

7.1.3 GLS Functional Coverage Evaluation and Reporting

This section defines how gate-level simulation (GLS) functional coverage results are evaluated and reported, as the results companion to the methodology in §6.10.2. In this flow, stimulus is software-driven: bare-metal C programs are compiled into RISC-V binaries, converted to VMEM images, and executed on the Simple System GLS platform. Therefore, coverage results represent behaviors reachable through the implemented memory map, peripherals, and control paths exposed by that platform.

After regression, coverage data from individual GLS runs is analyzed and merged in Cadence IMC to produce a single closure dataset. Evaluation is performed at covergroup and coverpoint/cross-bin granularity using three classes of bins:

1. Closed bins: bins hit during regression and counted as verified behaviors under gate-level conditions.

2. Unreachable bins: bins that cannot be exercised by construction in the current Simple System configuration and are formally excluded from achievable closure.

3. Reachable-open bins: bins that are architecturally and structurally reachable, but not yet hit by current VMEM tests; these are prioritized for follow-on stimulus development.

To avoid inflated or misleading percentages, closure is reported against the achievable space, not raw total bins. Achievable closure is computed as:

$$C_{\text{achievable}} = \frac{B_{\text{closed}}}{B_{\text{total}} - B_{\text{unreachable}}}.$$

This ensures that closure reflects what can actually be stimulated in this platform.

The reporting objective is to show that the implemented gate-level design is exercised across meaningful pipeline, interface, address-region, and MMIO behaviors; that exclusions are justified and traceable; and that remaining reachable-open bins are explicitly tracked as verification gaps with planned stimulus actions.

For documentation, coverage is exported from IMC after run merge into a normalized dataset containing, per covergroup: total bins, closed bins, reachable-open bins, and achievable closure percentage.

Table 7.2: *GLS Functional Coverage Closure Summary (Merged IMC Dataset)*

Covergroup	Closed	Total	Achiev. %	Notes / Primary Gap
cg_core_lite_if_data	29	33	87.88	Interface req/gnt/rvalid behavior largely covered; residual open bins are reachable handshake/response combinations.
cg_core_lite_addr_i	14	15	93.33	Region/address-class coverage is high; one remaining reachable-open bin persists.
cg_core_lite_stage	17	30	56.67	Lowest closure group; dominant remaining gaps are multi-stage flow/state combinations.
cg_core_lite_mmio_	61	65	93.85	MMIO and byte-enable access patterns are broadly exercised; few bins remain open.
cg_stage_flow + cg_lsu_irq	32	39	82.05	Aggregated from cg_stage_flow (24/31) and cg_lsu_irq (8/8).

The aggregate IMC total includes additional coverage objects under the parent monitor hierarchy beyond the covergroups listed in the closure table; therefore, the listed covergroup totals do not sum directly to the parent-scope bin count.

The complete Cadence IMC aggregation of coverage metrics results in us achieving $\frac{258}{294} = 87.76\%$ functional coverage, prior to any unreachability analysis being performed. Unreachability analysis was not attempted officially within the scope of this project, despite some bins being known to be unreachable, due to the lack of time for any formal analysis on

the baseline RTL of the Ibex core. A value above 80%, in most cases of verification, can be deemed as acceptable, however in future work we would like to see this value improved upon to ensure correctness and secure functionality of our design.

7.2 Validation of Post-RTL Security Integrity

This section explains various testing methods that can be used on the Ibex core before and after adding our extension. The main method used would be to collect measurements that specifically show changes in leakage rather than a pass/fail test on functionality. Since we will be using ASIC design for our project, we plan to use simulation-based power analysis to test our extension. With simulations, we will be able to accurately configure and control our experimentation on the design. Simulation provides an environment for comparison before and after our modifications without using a hardware-based environment. We will first characterize the unmodified Ibex, then repeat the same procedures after modifying the core, using identical firmware and capture settings for fair comparison.

7.3 Security Sign-off Verification

7.3.1 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, 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 accurately re-run analysis. Each trace file will be stored with a small metadata record such as board, bitstream hash, firmware hash, clock, shunt value, sample rate, trigger offsets, plaintext list, and 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 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.

7.4 Comparative SCA Methodology

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 modifications made to the core.

7.4.1 Baseline Characterization and Attacker Model

Attacker Model

AES-128 encryption on an Ibex-based system serves as the target workload. Each simulation run fixes the secret key and uses a known plaintext schedule. Testbench stimulus raises a trigger signal immediately before the round sequence to align simulated power traces to the same operation window, and one encryption corresponds to one short trace segment. The attacker is assumed to know the plaintexts and observe the power trace per encryption while not knowing the key, which is standard for embedded SCA evaluation and directly compatible with SPA/DPA/CPA analysis [79].

Capture Setup

To ensure measurement consistency, the simulation testbench executes AES-128 with a fixed key per session and a pre-defined list of plaintexts. The firmware asserts a specific GPIO signal immediately before the AES rounds begin, and this signal acts as a trigger event for the EDA tools to synchronize the start of the power estimation window for every encryption. Each encryption operation corresponds to a distinct power trace segment generated from the switching activity. We begin with a short pilot simulation to optimize the sampling resolution (time step), verify the trigger latency, and define the recording window size. This pilot phase ensures that the full AES execution is captured within the VCD dump, allowing Simple Power Analysis (SPA) to reveal the expected repeatable round structure before we batch-process the large dataset for statistical evaluation.

Baseline Testing Procedure

This phase establishes the baseline leakage behavior of the unmodified Ibex core for meaningful comparisons against the post-modification core. This plan covers functional testing, power trace generation through gate-level simulation during AES encryptions, timing sanity checks, and data logging for repeatability. The same firmware, trigger signal logic, clock frequency, and simulation capture windows used here are reused after the modification so differences can be attributed strictly to the extension logic rather than the testbench setup.

The simulation testbench executes a small firmware application that repeatedly encrypts known plaintexts under a fixed key per session. A dedicated GPIO signal is toggled in the RTL immediately around the AES round sequence to serve as a synchronization marker for the power estimation tool. Before generating large datasets, we confirm correct AES outputs against known golden values, verify that the trigger signal transitions at the correct clock cycle, and ensure that the targeted round activity falls entirely within the simulation dump window. A short pilot simulation is performed to validate the VCD sampling interval and define precise start and stop times relative to the trigger to optimize storage efficiency.

Power Trace Capture

With alignment confirmed, gate-level simulation with annotated delays collects thousands of power trace samples using Cadence Xcelium's power estimation features. The capture conditions are kept constant across runs, with the same clock frequency, trigger alignment, and AES firmware loop. A quick SPA overlay verifies that the trace features are repeatable enough for downstream DPA/CPA analysis. The larger dataset is then exported together with a compact metadata file.

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.

7.4.2 Leakage Models and Technical Rationale

The primary point of interest is the first-round S-box, which is both sensitive and well documented in side-channel literature [80, 81]. A Hamming-weight model of the S-box output provides the initial leakage hypothesis, while 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 our modifications introduces intentional desynchronization through clock randomization, the window is widened and light software re-alignment is applied. Consistency of windows and models across "before" and "after" runs is maintained to keep comparisons fair.

Hamming Weight vs. Other Leakage Models

The Hamming Weight (HW) model is selected as our primary leakage hypothesis because the analysis targets the first AES S-box output, where data is freshly computed and loaded into registers with no deterministic prior state [80]. Additionally, in CMOS technology, dynamic power from data-dependent capacitance charging dominates, which makes Hamming Weight a reliable proxy for power consumption in simulation [81].

Table 7.3: *Side-Channel Leakage Model Comparison*

Model	Target Mechanism	Key Assumption	Strengths	Weaknesses
Hamming Weight (HW)	State Value (Static)	Power consumption is proportional to the number of bits set to '1'.	Simplest to implement, requires no knowledge of the previous state, good for S-box output.	Physically inaccurate for CMOS registers, lower correlation.
Hamming Distance (HD)	State Transition (Dynamic)	Power consumption is proportional to the number of bit flips ($0 \rightarrow 1$ or $1 \rightarrow 0$).	Physically accurate for CMOS logic, highly correlated to ASIC power, standard for register updates.	Requires a detailed profiling phase to determine weights (w), difficult to apply in simulation without back-annotation.

In Table 7.3, we compare the differences between the Hamming Weight Model and the Hamming Distance Model, which further demonstrates how the HW model works better for our project. While the table notes that Hamming Distance is physically more accurate for CMOS logic since it accounts for specific bit transitions, its main drawback is that it requires precise knowledge of the previous state. However, in our case of AES, the registers often hold indeterminate data before the S-box output is written and it ends up simplified to zero. Through selecting the Hamming Weight model, we can take advantage of its strengths to effectively identify leakage without the complexity of tracking every register transition in the simulation.

7.4.3 Statistical Procedures and Controls

Attack Procedures

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 our modified core to avoid analysis bias.

Control Conditions & Data Handling

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 computational budget required to recover the key, or preventing stable peaks from forming under identical simulation parameters. Observable changes in the simulated power signatures are noted as supporting evidence.

Conditions are held constant across datasets: same testbench configuration, monitored netlist hierarchy, clock frequency, trigger logic, simulation dump window, plaintext schedule, and analysis scripts. Each run carries a compact metadata record (netlist version, firmware hash, clock period, simulator time-step resolution, simulation window, and trace counts) to support exact repetition. If any stability fix requires changing the simulation window or sampling resolution, the corresponding baseline is re-simulated so comparisons remain matched.

Outcome statements emphasize theoretical resistance validated through simulation. If early correlation peaks observed on the baseline simulation do not appear under the same trace count after secure hardware integration, 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 our modified core's intended mechanism to connect power estimation results with design intent.

7.5 Physical Validation Plan (Stretch Goal)

Post-modification testing evaluates leakage behavior after integrating the secure hardware modifications into Ibex. Throughout the design flow, side-channel checkpoints provide visibility into the impact of the modifications: RTL simulation ensures functional correctness and timing sanity, gate-level power estimates characterize switching trends, and post-layout reports quantify area, timing, and power overhead up to GDSII. These stages collectively establish a verification narrative around implementation cost, structural changes, and whether simulation-based power signatures indicate reduced side-channel leakage.

As a potential extension to simulation-based evaluation, the modified RISC-V core could be hosted on an FPGA and the same SPA/DPA/CPA attack procedures used for the baseline may be repeated in hardware. This optional physical validation step would mirror the original configuration, including identical trigger protocols, clock conditions, and power measurement methods such as oscilloscope-based voltage observation across a series sense resistor. If pursued, such FPGA captures would serve as a practical proxy for ASIC behavior and provide an additional data point confirming whether leakage reductions observed in simulation translate into real-world traces.

7.5.1 FPGA Implementation

To further test power-based attacks on the Ibex core, we would 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 additional hardware, 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.

We would implement the basic Ibex system on a Basys 3 FPGA, adding the necessary components to allow the program to run locally and generate a trigger signal. The firmware will use this trigger to alert the test equipment, perform an encryption, and then repeat

the process with a list of known messages. We will select a moderate clock speed that allows the oscilloscope to capture clean signals without interference. To measure power consumption, we will place a small resistor in the board's main power supply line. As the processor operates, the changes in current will create small voltage fluctuations across this resistor, which we will record using a probe.

7.5.2 Main Experimental Controls

To properly mitigate noise and ensure the experiment can be reproduced through FPGA implementation, we would have to establish strict controls such as:

- Stable Clock: We will run the board at a fixed, lower frequency to minimize high-frequency noise.
- Temperature Stability: The board will be allowed to warm up for a set period before recording begins to ensure thermal consistency.
- Consistent Data: We will use the exact same schedule of input messages (plaintexts) as used in our computer simulations.
- Detailed Logging: We will record all environmental details and hardware identifiers.

The setup will first be validated with a short pilot run to check that the trigger is clean and the encryption happens within the capture window. Only after these checks are confirmed will the full data collection proceed.

7.5.3 Expected Outcomes

If this physical validation were to be performed, we would expect the unprotected Ibex on the Basys 3 to reveal the key with a relatively small number of measurements, which is typical for this hardware. For the modified core, our target is to see a massive increase in the number of measurements required to find the key, or ideally, to see no leakage at all within a reasonable timeframe. It is important to note that FPGAs are inherently noisier than the final custom chips we are designing, so the simulation analysis in the previous sections remains the primary evidence for our security claims.

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 SCA resistant processor, 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 the NG RAD project. 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

Table 8.3: *Senior Design 1 Timeline*

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 (NG RAD), 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 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)
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

Table 8.4: *Senior Design 2 Timeline*

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

The projected work distribution has been planned so each portion has a primary assignee, and 1-2 secondary assignees. While collaborating with our sponsor, we delegated tasks accordingly, but left room for each individual to explore all areas of the project if desired. Much of the delegation will happen on a per-semester basis (differing between SD1 and SD2) due to the unique natures of front-end and back-end VLSI.

Table 8.6: *Work Distribution*

Work Distribution		
Primary	**Secondary**	**Task(s)**
Anna Craig	Alyssa Pinnock	Verification of program counter, instruction cache, and prefetch buffer
Alexander DeFalco	Anna Craig	Design of custom memory structure and Data Memory Interface verification
Joshua Joseph	Daniel Odi	Modified Core design, decoder, CSR, and multiplication handler verification
Daniel Odi	Joshua Joseph	FIFO, ALU, and Load-Store Unit verification
Alyssa Pinnock	Alexander DeFalco / Anna Craig	Control unit, Instruction Memory Interface, and Compression Instruction Decoder verification

8.3.4 Project Milestone Summary

The project milestones for NG RAD are structured 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 complete verification of the custom 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, validate engineering requirements, complete any stretch goals as fit.

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

In conclusion, the NG RAD project allows us to gain more experience in the field of ASIC design and makes us better equipped with the proper knowledge needed in the semiconductor industry. Through following the full-flow ASIC design process, we will be able to create an extension on a RISC-V core that aims to reduce side-channel leakage and prevent power-based side-channel attacks. The process described in this proposal will assist us in achieving this goal, and provides a detailed overview of our implementation.

Beyond the technical challenge of learning new tools, this project is able to offer a valuable opportunity to bridge the gap between limited coursework in ASIC design and the expectations of the industry. By working with industry-grade Cadence EDA tools and applying the structured full-flow design process, we will gain realistic experience compared to what is expected in modern semiconductor workflows. Utilizing the Ibex RISC-V core provides us a strong foundation to extend the design and implement additional security features to mitigate side-channel attacks.

As our project transitions to the next phase, we will shift our focus to refining our verification and validation techniques of the design modules to thoroughly test our design and ensure it works as intended. Our planned next steps are test-oriented and focus on continuing thorough testing of the design. Through verifying the design as a team, we are able to actively follow the ASIC design process and earn industry-like experience of working on a design team. In Senior Design II, we plan to expand coverage for the custom modules, complete gate-level synthesis and run post-synthesis equivalence and masking checks. If time allows, we can also implement our stretch goal of deciding physical testing implementations to preserve intended leakage reduction needs while meeting STA and DRC/LVS requirements. Once we analyze the power and timing behavior, we then will prepare the design for a mock-GDSII tapeout. Achieving these deliverables with not only demonstrate functional correctness of the design, but also that our modified core can be developed through a realistic fabrication workflow.

Ultimately, NG RAD combines targeted hardware security techniques and attack countermeasures, established verification methods, and a full ASIC design tool flow to evaluate whether side-channel resilient designs at the RTL level can be delivered to tapeout through synthesis and layout. The deliverables and procedures documented here form a practical roadmap to Senior Design II to produce a gate-level implementation, validate both functional and technical requirements, and provide quantitative evidence on effectiveness and cost of the design. Successfully completing these tasks will demonstrate that our approach is both implementable within a realistic design flow and instructive as an educational bridge between academic study and industry expectations in secure ASIC design.

Bibliography

[1] 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.

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

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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).

[11] 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.

[12] 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.

[13] 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.

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

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

[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] N. I. of Standards and Technology, "Nist sp 800-90b: Recommendation for the entropy sources used for random bit generation." https://csrc.nist.gov/pubs/sp/800/90/b/final, 2018. Accessed: Nov. 18, 2025.

[29] N. I. of Standards and Technology, "Nist sp 800-90a rev.1: Recommendation for random number generation using deterministic random bit generators." https://csrc.nist.gov/pubs/sp/800/90/a/r1/final, 2015. Accessed: Nov. 18, 2025.

[30] M. Rosulek, "The joy of cryptography." https://joyofcryptography.com/, 2021. Draft textbook, Accessed: Nov. 22, 2025.

[31] N. I. of Standards and Technology, "Nist sp 800-22 rev.1a: A statistical test suite for random and pseudorandom number generators for cryptographic applications." https://csrc.nist.gov/pubs/sp/800/22/r1/upd1/final, 2010. Accessed: Nov. 18, 2025.

[32] T. I. Incorporated, "What's an lfsr?," Tech. Rep. SCTA036A, Texas Instruments Incorporated, Dec. 1996. Accessed: Nov.25,2025.

[33] N. H. E. Weste and D. M. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley, 4th ed., 2011. Online version (PDF) accessed: Nov. 25, 2025 — https://vlsiworlds.com/wp-content/uploads/2021/04/cmos-vlsi-design-by-weste.pdf.

[34] L. Guangxi, "Gng specification 1.1." https://github.com/liuguangxi/gng/blob/master/doc/gng_spec_1.1.pdf, 2025. Version 1.1, accessed: Nov. 12, 2025.

[35] E. B. Smid, S. Leigh, M. Levenson, M. Vangel, A. DavidBanks, and S. JamesDray, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," *Her research interest includes Computer security, secure operating systems, Access control, Distributed systems, Intrusion detection systems*, 2010.

[36] S. Picek, B. Yang, V. Rozic, J. Vliegen, J. Winderickx, T. De Cnudde, and N. Mentens, "Prngs for masking applications and their mapping to evolvable hardware," in *Smart Card Research and Advanced Applications* (K. Lemke-Rust and M. Tunstall, eds.), (Cham), pp. 209–227, Springer International Publishing, 2017.

[37] L. G. de la Fraga, J. D. Rodríguez-Muñoz, and E. Tlelo-Cuautle, *Statistical Tests for PRNGs*, pp. 83–93. Cham: Springer Nature Switzerland, 2025.

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

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

[40] 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.

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

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

[43] Unknown, "Presentation paper no. 04 at the workshop on secure risc-v architecture design (secrisc-v'20)," in *Proceedings of the* Workshop on Secure RISC-V Architecture Design (SECRISC-V'20), 2020. Accessed: Nov. 25, 2025.

[44] Z. He, X. Deng, B. Yang, K. Dai, and X. Zou, "A sca-resistant processor architecture based on random delay insertion," in *2015 International Conference on Computing and Communications Technologies (ICCCT)*, pp. 278–281, 2015.

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

[46] SkyWater Technology Foundry, "Skywater sky130 pdk — digital standard cell libraries." https://skywater-pdk.readthedocs.io/en/main/contents/libraries.html#foundry-provided-digital-standard-cell-libraries, 2025. Accessed: Nov. 25, 2025.

[47] GlobalFoundries & Google, "Gf180mcu pdk — digital libraries." https://gf180mcu-pdk.readthedocs.io/en/latest/digital/Digital.html, 2025. Accessed: Nov. 25, 2025.

[48] C. D. Systems, *GPDK045 Reference Manual: Generic 45 nm Salicide 1.0 V/1.8 V 1P/11M Process Design Kit and Rule Decks (PRD), Revision 6.0*. Cadence Design Systems, 2019. Accessed: Nov. 25, 2025.

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

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

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

[56] O. Kindgren, "Fusesoc user guide." https://fusesoc.readthedocs.io/en/stable/user/, 2025. Accessed: Nov.242025.

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

[58] pulp-platform, "bender: A dependency management tool for hardware projects." https://github.com/pulp-platform/bender, 2025. Accessed: Nov.242025.

[59] Chipyard, "Welcome to chipyard's documentation (version 1.11.0)." https://chipyard.readthedocs.io/en/stable/index.html. Chipyard Documentation, Accessed: Sep. 4, 2025.

[60] National Institute of Standards and Technology, "Fips 140-2 security requirements for cryptographic modules." https://csrc.nist.gov/pubs/fips/140-2/upd2/final, 2001. Accessed: Nov. 18, 2025.

[61] National Institute of Standards and Technology, "FIPS PUB 197: Advanced encryption standard (aes)." https://csrc.nist.gov/pubs/fips/197/final, 2001. U.S. Department of Commerce.

[62] I. of Electrical and E. Engineers, "Ieee p1363: Standard specifications for public key cryptography." file:///mnt/data/p1363.pdf, 2000. Accessed: Nov. 22 2025.

[63] A. B. Luís T. M. Davidson, and A. Vassilev, "Nist ir 8214a: Roadmap toward criteria for threshold schemes for cryptographic primitives." https://csrc.nist.gov/pubs/ir/8214/a/final, 2020. Accessed: Nov. 21 2025.

[64] I. of Electrical and E. Engineers, "Ieee 1800-2017: Standard for systemverilog — unified hardware design, specification, and verification language." file:///mnt/data/1800-2017.pdf, 2017. Accessed: Nov. 23 2025.

[65] C. D. Systems, "Timing closure." file:///mnt/data/cadence_timing_closure.html, 2025. Accessed: Nov. 20 2025.

[66] E. Corpeño, "An introduction to risc-v — understanding risc's open isa." All About Circuits Technical Article, June 2022. Accessed: Nov.242025.

[67] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v compressed instruction set manual, version 1.9." UC Berkeley Technical Report UCB/EECS-2015-209, 2015.

[68] C. Lu, "Dynamic linking in trusted execution environments in risc-v." UC Berkeley Technical Report EECS-2022-142, 2022.

[69] A. Waterman, K. Asanović, and J. Hauser, "The risc-v instruction set manual, volume ii: Privileged architecture." RISC-V International, Document Version 20211203, 2021.

[70] 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.

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

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

[73] O. Project, "Hardware development stages (hardware verification stages v)." https://opentitan.org/book/doc/project_governance/development_stages.html#hardware-verification-stages-v. OpenTitan Documentation, Accessed: Oct. 21, 2025.

[74] lowRISC Ltd., "Instruction decode and execute – ibex reference guide." https://ibex-core.readthedocs.io/en/latest/03_reference/instruction_decode_execute.html, 2025. Accessed: Nov. 20 2025.

[75] D. . Reuse, "Verification and generation of constraints." https://www.design-reuse.com/article/59424-verification-and-generation-of-constraints/, 2009. Accessed: Nov. 18, 2025.

[76] lowRISC, "lowrisc risc-v gcc toolchain release 20220210-1." https://github.com/lowRISC/lowrisc-toolchains/releases/tag/20220210-1, 2022. Accessed: Nov. 24, 2025.

[77] I. Cadence Design Systems, "Conformal equivalence checker." https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/logic-equivalence-checking/conformal-equivalence-checker.html. Accessed: Nov. 6, 2025.

[78] I. Cadence Design Systems, "Cadence rtl-to-gdsii flow training." https://www.cadence.com/en_US/home/training/all-courses/86136.html, 2025. Accessed: Nov. 24, 2025.

[79] F.-X. Standaert, T. G. Malkin, and M. Yung, "A unified framework for the analysis of side-channel key recovery attacks," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 443–461, Springer, 2009.

[80] É. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Workshop on Cryptographic Hardware and Embedded Systems*, 2004.

[81] L. Tebelmann, M. Pehl, and G. Sigl, "Em side-channel analysis of bch-based error correction for puf-based key generation," in *Proceedings of the 2017 Workshop on Attacks and Solutions in Hardware Security*, ASHES '17, (New York, NY, USA), p. 43–52, Association for Computing Machinery, 2017.

Chapter A - Software Code

A.1 Module Designs (Verilog)

A.1.1 casr37.v

```
module casr37(input  wire        clk,
              input  wire        rst_n,
              input  wire        i_en,
              input  wire        i_ptb,
              input  wire        i_ptb_valid,
              output reg  [36:0] o_state);

  wire [36:0] nxt_state;

  integer idx;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      o_state <= 37'b0;
    end else begin
      if (i_en)
        for (idx = 0; idx < 37; idx = idx+1)
          o_state[idx] <= nxt_state[idx];
      else begin
          o_state <= o_state;
      end
    end

  genvar  i;

  generate
    // From: Cattell 1995, JET, matches the model in `digital/python/casr.py`
    for (i = 1; i <= 37; i = i+1) begin : casr_blk
      wire left  = (i > 1)  ? o_state[37-i+1] : i_ptb & i_ptb_valid; // perturbed instea
      wire right = (i < 37) ? o_state[37-i-1] : i_ptb & i_ptb_valid; // perturbed instea
      assign nxt_state[37-i] = (i == 9) ? left ^ right ^ o_state[37-9] : left ^ right;
    end
  endgenerate
```

```
endmodule
```

A.1.2 clkgen.v

```
‘timescale 1ns/1ps
module clkgen(
    input  wire i_clk,
    input  wire rst_n,
    output reg  o_clk
);

    wire [7:0] lfsrOutput8;
    wire [1:0] selectLines;
    wire p25, p50, p75;
    reg  skip_clk;
    reg  skip_clk_r;  // Registered version of skip_clk for stability

    wire ptb_bit;
    wire ptb_bit_valid;

    // Instantiate 8-bit LFSR
    lfsr8 generateClockSkip(
        .clk(i_clk),
        .rst_n(rst_n),
        .i_ptb(ptb_bit),
        .i_ptb_valid(ptb_bit_valid),
        .o_state(lfsrOutput8)
    );

    lfsr43 generatePerturbation(
        .clk(i_clk),
        .rst_n(rst_n),
        .o_ptb(ptb_bit),
        .o_ptb_valid(ptb_bit_valid)
    );

    // Output signals for 25%, 50%, 75%
    assign p25 = lfsrOutput8[7] & lfsrOutput8[6];
    assign p50 = lfsrOutput8[7] ^ lfsrOutput8[6];
    assign p75 = lfsrOutput8[7] | lfsrOutput8[6];

    // Select lines based on parity of subsets of LFSR bits
    assign selectLines = {^lfsrOutput8[2:0], ^lfsrOutput8[5:3]};

    // Clock skip selection logic
    always @* begin
```

```
        case (selectLines)
            2'b00:   skip_clk = 1'b0;
            2'b01:   skip_clk = p25;
            2'b10:   skip_clk = p50;
            2'b11:   skip_clk = p75;
            default: skip_clk = 1'b0;
        endcase
    end

    // Register skip_clk to avoid race conditions with i_clk
    always @(posedge i_clk or negedge rst_n) begin
        if (!rst_n)
            skip_clk_r <= 1'b0;
        else
            skip_clk_r <= skip_clk;
    end

    // Main output clock generation
    always @(posedge i_clk or negedge rst_n) begin
      if (!rst_n) begin
            o_clk <= 1'b0;
      end else if (~skip_clk_r) begin
            $display("Clock edge skipped");
            o_clk <= ~o_clk; // Hold value (skip edge)
      end else begin
            o_clk <= o_clk;
      end
    end
endmodule
```

A.1.3 lfsr8.v

```
// ------------------------------------------------------------------------
// 8-bit XNOR-based LFSR
// ------------------------------------------------------------------------
module lfsr8(
    input  wire       clk,
    input  wire       rst_n,
    input  wire       i_ptb,
    input  wire       i_ptb_valid,
    output reg [7:0]  o_state
);
    // Feedback taps for polynomial x^8 + x^6 + x^5 + x^4 + 1
    // Using XNOR instead of XOR
    wire feedback;
    assign feedback = ~(o_state[7] ^ o_state[5] ^ o_state[4] ^ o_state[3]) ^
                      (i_ptb & i_ptb_valid);
```

```
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            o_state <= 8'b01001100; // Non-zero seed
        else
            o_state <= {o_state[6:0], feedback};
    end

endmodule
```

A.1.4 lfsr43.v

```
module lfsr43 (
    input  wire        clk,
    input  wire        rst_n,
    output reg         o_ptb,
    output reg         o_ptb_valid
);

  wire feedback;
  wire max_pulse_counter;
  reg  [42:0] o_state;

  // Example taps for a 43-bit maximal LFSR
  assign feedback = ~(o_state[42] ^
                      o_state[41] ^
                      o_state[37] ^
                      o_state[36]);

  counter_5bit ptbCounter (
    .clk(clk),
    .rst_n(rst_n),
    .max_pulse(max_pulse_counter)
  );

  always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        o_state      <= 43'h1ABCDE12345; // non-zero seed
        o_ptb        <= 1'b0;
        o_ptb_valid  <= 1'b0;
    end else begin
        o_state      <= {o_state[41:0], feedback};
        o_ptb        <= ^o_state[27:21];
        o_ptb_valid  <= max_pulse_counter;
    end
  end
endmodule
```

A.1.5 counter_5bit.v

```
module counter_5bit (
    input  wire clk,
    input  wire rst_n,
    output reg  max_pulse
);
    reg [4:0] count;

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            count     <= 5'd0;
            max_pulse <= 1'b0;
        end else begin
            if (count == 5'd31) begin
                count     <= 5'd0;
                max_pulse <= 1'b1;
            end else begin
                count     <= count + 1'b1;
                max_pulse <= 1'b0;
            end
        end
    end
endmodule
```

A.1.6 ring_oscillator.v

```
module ring_oscillator #(
    parameter N = 5  // number of stages, must be odd
)(
    input wire clk,
    input wire enable,   // enables the oscillator
    input wire rst_n     // synchronous reset
);

    // Internal stages of the ring
    reg [N-1:0] stages;

    integer i;

    always @(posedge rst_n or posedge clk) begin
        if (!rst_n) begin
            stages <= {N{1'b0}};
        end else if (enable) begin
            // Toggle stages to form a ring oscillator
            for (i = 0; i < N; i = i + 1) begin
                if (i == 0)
                    stages[i] <= ~stages[N-1];  // first stage inverts last stage
```

```
                else
                    stages[i] <= ~stages[i-1];  // other stages invert previous stage
            end
        end
    end

    // Use stages internally so synthesis does not remove them
    wire unused = |stages;
endmodule
```

A.2 Testbench Designs (Verilog)

A.2.1 casr37_tb.v

```
`timescale 1ns/1ps

module casr37_tb;

  // DUT inputs
  reg        clk;
  reg        rst_n;
  reg        i_en;
  reg        i_ptb;
  reg        i_ptb_valid;

  // DUT outputs
  wire [36:0] o_state;

  // Loop variable
  integer k;

  // DUT instantiation
  casr37 dut (
    .clk         (clk),
    .rst_n       (rst_n),
    .i_en        (i_en),
    .i_ptb       (i_ptb),
    .i_ptb_valid (i_ptb_valid),
    .o_state     (o_state)
  );

  // 100 MHz clock (10 ns period)
  initial clk = 1'b0;
  always #5 clk = ~clk;

  initial begin
    // Waveform dump
```

```
$dumpfile("casr37_tb.vcd");
$dumpvars(0, casr37_tb);

// Initial conditions
rst_n       = 1'b0;
i_en        = 1'b0;
i_ptb       = 1'b0;
i_ptb_valid = 1'b0;

// Release reset after a few cycles
repeat (3) @(posedge clk);
rst_n = 1'b1;

// ------------------------------------------------------------
// Phase 1: CA disabled (i_en = 0) | state should remain all zero
// ------------------------------------------------------------
repeat (8) @(posedge clk);

// ------------------------------------------------------------
// Phase 2: Enable CASR and inject a short perturbation
// First 4 cycles: i_ptb = 1, i_ptb_valid = 1 (seed from both ends)
// Remaining cycles: pure CA evolution (no new perturbations)
// ------------------------------------------------------------
i_en = 1'b1;

for (k = 0; k < 16; k = k + 1) begin
  @(posedge clk);
  if (k < 4) begin
    i_ptb       = 1'b1;
    i_ptb_valid = 1'b1;
  end else begin
    i_ptb       = 1'b0;
    i_ptb_valid = 1'b0;
  end
end

// ------------------------------------------------------------
// Phase 3: Hold state (i_en = 0) while toggling inputs
// ------------------------------------------------------------
i_en = 1'b0;
for (k = 0; k < 8; k = k + 1) begin
  @(posedge clk);
  i_ptb       = ~i_ptb;
  i_ptb_valid = k[0];  // arbitrary activity on inputs
end

// ------------------------------------------------------------
```

```
    // Phase 4: Re-enable CASR with occasional random perturbations
    // ------------------------------------------------------------
    i_en = 1'b1;
    for (k = 0; k < 16; k = k + 1) begin
      @(posedge clk);
      i_ptb       = $random;
      i_ptb_valid = (k % 3 == 0);
    end

    @(posedge clk);
    $finish;
  end

endmodule
```

A.2.2 tb_clkgen.v

```
`timescale 1ns/1ps
module tb_clkgen;
    reg clk;
    reg fakeClk;
    reg rst_n;
    wire outputClk;

    // DUT instance
    clkgen clock(
        .i_clk(clk),
        .rst_n(rst_n),
        .o_clk(outputClk)
    );

    // Clock generation: 100 MHz (10 ns period)
    initial clk = 0;
    initial #10 fakeClk = 0;
    always #5  clk     = ~clk;
    always #10 fakeClk = ~fakeClk;

    // Reset and simulation control
    initial begin
        // Create waveform dump file for GTKWave
        $dumpfile("clkgen_tb.vcd");   // name of the waveform file
        $dumpvars(0, tb_clkgen);      // dump all signals in this testbench hierarchy

        // Reset sequence
        rst_n = 0;
        #15 rst_n = 1;
        $display("Reset complete at time %0t", $time);
```

```
        // Run simulation for some time
        #6000;
        $display("Simulation finished at time %0t", $time);
        $finish;
    end

    // Monitor output every positive and negative clock edge
    always @(posedge clk or negedge clk)
        $display("time=%0t : outputClk=%b when clk=%b",
                 $time, outputClk, clk);

endmodule
```

A.2.3 lfsr8_tb.v

```
‘timescale 1ns/1ps

module lfsr8_tb;

  // DUT inputs
  reg        clk;
  reg        rst_n;
  reg        i_ptb;
  reg        i_ptb_valid;

  // DUT outputs
  wire [7:0] o_state;

  // Loop variable (declare at module scope to avoid block-decl issues)
  integer k;

  // DUT instantiation
  lfsr8 dut (
    .clk        (clk),
    .rst_n      (rst_n),
    .i_ptb      (i_ptb),
    .i_ptb_valid(i_ptb_valid),
    .o_state    (o_state)
  );

  // ------------------------------------------------------------------------
  // Clock generation: 100 MHz (10 ns period)
  // ------------------------------------------------------------------------
  initial clk = 1’b0;
  always #5 clk = ~clk;
```

```
// ------------------------------------------------------------------------
// Stimulus
// ------------------------------------------------------------------------
initial begin
  // Waveform dump
  $dumpfile("lfsr8_tb.vcd");
  $dumpvars(0, lfsr8_tb);

  // Initial conditions
  rst_n       = 1'b0;
  i_ptb       = 1'b0;
  i_ptb_valid = 1'b0;

  // Hold reset low for a few cycles
  repeat (3) @(posedge clk);
  rst_n = 1'b1;

  // ------------------------------------------------------------------------
  // Phase 1: Pure LFSR behavior (no external mixing)
  // i_ptb_valid = 0 so feedback is driven only by XNOR taps.
  // ------------------------------------------------------------------------
  repeat (16) begin
    @(posedge clk);
    i_ptb       = 1'b0;
    i_ptb_valid = 1'b0;
  end

  // ------------------------------------------------------------------------
  // Phase 2: Occasional mixing of i_ptb when i_ptb_valid is asserted.
  // Every 4th cycle, toggle i_ptb and assert i_ptb_valid.
  // ------------------------------------------------------------------------
  i_ptb = 1'b0;
  for (k = 0; k < 32; k = k + 1) begin
    @(posedge clk);
    if ((k % 4) == 0) begin
      i_ptb       = ~i_ptb;
      i_ptb_valid = 1'b1;
    end else begin
      i_ptb       = 1'b0;
      i_ptb_valid = 1'b0;
    end
  end

  // ------------------------------------------------------------------------
  // Phase 3: Continuous mixing (stress behavior)
  // ------------------------------------------------------------------------
  repeat (16) begin
```

```
      @(posedge clk);
      i_ptb       = $random;
      i_ptb_valid = 1'b1;
    end

    // End simulation
    @(posedge clk);
    $finish;
  end

endmodule
```

A.2.4 tb_lfsr43.v

```
// Code your testbench here
// or browse Examples
`timescale 1ns/1ps

module tb_lfsr43;

  reg clk;
  reg rst_n;
  wire o_ptb;
  wire o_ptb_valid;

  // DUT
  lfsr43 dut (
    .clk(clk),
    .rst_n(rst_n),
    .o_ptb(o_ptb),
    .o_ptb_valid(o_ptb_valid)
  );

  // Clock: 10ns period
  always #5 clk = ~clk;

  initial begin
    // This enables waveform generation in EDA Playground
    $dumpfile("dump.vcd");
    $dumpvars(0, tb_lfsr43);

    clk   = 0;
    rst_n = 0;

    #20 rst_n = 1;

    repeat (200) @(posedge clk);
```

```
    $finish;
  end

endmodule
```

Chapter B - GLS Program Sources (C)

This appendix captures the C sources used to generate VMEM images for gate-level simulation in the Simple System flow. The program set below corresponds to the default test compilation flow for GLS and includes the shared support source linked into each program.

B.0.1 Shared Support: simple_system_common.c

B.0.2 bubblesort_test.c

```
// Bubble sort test - sorts array and verifies result
#include "simple_system_common.h"

void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
            }
        }
    }
}

int main(void) {
    puts("=== Bubble Sort Test ===\n");

    // Use volatile writes to avoid memcpy optimization
    volatile int arr[20];
    arr[0]=64; arr[1]=-12; arr[2]=25; arr[3]=0; arr[4]=22;
    arr[5]=-7; arr[6]=11; arr[7]=90; arr[8]=-33; arr[9]=1;
    arr[10]=55; arr[11]=-88; arr[12]=42; arr[13]=17; arr[14]=-5;
    arr[15]=100; arr[16]=3; arr[17]=-1; arr[18]=77; arr[19]=8;

    int expected[20];
    expected[0]=-88; expected[1]=-33; expected[2]=-12; expected[3]=-7; expected[4]=-5;
    expected[5]=-1; expected[6]=0; expected[7]=1; expected[8]=3; expected[9]=8;
    expected[10]=11; expected[11]=17; expected[12]=22; expected[13]=25; expected[14]=42;
    expected[15]=55; expected[16]=64; expected[17]=77; expected[18]=90; expected[19]=100;

    // Copy to non-volatile for sorting
    int sort_arr[20];
    for (int i = 0; i < 20; i++) sort_arr[i] = arr[i];
    int n = 20;

    bubble_sort(sort_arr, n);
```

```
    int pass = 1;
    for (int i = 0; i < n; i++) {
        if (sort_arr[i] != expected[i]) {
            puts("FAIL at index ");
            puthex(i);
            puts(": got ");
            puthex(sort_arr[i]);
            puts(" expected ");
            puthex(expected[i]);
            putchar('\n');
            pass = 0;
        }
    }

    // Print sorted array
    puts("Sorted: ");
    for (int i = 0; i < n; i++) {
        puthex(sort_arr[i]);
        putchar(' ');
    }
    putchar('\n');

    if (pass) {
        puts("PASS: Array correctly sorted!\n");
    } else {
        puts("FAIL: Sort errors detected\n");
    }

    sim_halt();
    return 0;
}
```

B.0.3 crc32_test.c

```
// CRC32 test - computes CRC32 of "123456789" and checks result
#include "simple_system_common.h"

static unsigned int crc32_table[256];

void crc32_init(void) {
    for (unsigned int i = 0; i < 256; i++) {
        unsigned int crc = i;
        for (int j = 0; j < 8; j++) {
            if (crc & 1)
                crc = (crc >> 1) ^ 0xEDB88320;
            else
                crc >>= 1;
        }
        crc32_table[i] = crc;
    }
}

unsigned int crc32(const char *data, int len) {
    unsigned int crc = 0xFFFFFFFF;
    for (int i = 0; i < len; i++) {
        unsigned char byte = (unsigned char)data[i];
        crc = (crc >> 8) ^ crc32_table[(crc ^ byte) & 0xFF];
    }
    return ~crc;
}

int main(void) {
    puts("CRC32 Test\n");

    crc32_init();
```

```
    unsigned int result = crc32("123456789", 9);

    puts("CRC32 of '123456789': 0x");
    puthex(result);
    putchar('\n');

    if (result == 0xCBF43926) {
        puts("PASS: CRC32 correct!\n");
    } else {
        puts("FAIL: expected 0xCBF43926\n");
    }

    sim_halt();
    return 0;
}
```

B.0.4 fibonacci_quick_demo_test.c

```
// Fibonacci quick demo test - prints fib(1) through fib(5), and fib(19)
#include "simple_system_common.h"

static void putdec(unsigned int value) {
  char buf[10];
  int pos = 0;

  if (value == 0) {
    putchar('0');
    return;
  }

  while (value > 0 && pos < (int)sizeof(buf)) {
    buf[pos++] = (char)('0' + (value % 10));
    value /= 10;
  }

  while (pos > 0) {
    putchar(buf[--pos]);
  }
}

int main(void) {
  puts("=== Fibonacci Quick Demo Test ===\n");
  int pass = 1;

  volatile unsigned int expected[20];
  expected[0] = 0;    expected[1] = 1;    expected[2] = 1;    expected[3] = 2;    expected[4] = 3;
  expected[5] = 5;    expected[6] = 8;    expected[7] = 13;   expected[8] = 21;   expected[9] = 34;
  expected[10] = 55;  expected[11] = 89;  expected[12] = 144; expected[13] = 233; expected[14] = 377;
  expected[15] = 610; expected[16] = 987; expected[17] = 1597; expected[18] = 2584; expected[19] = 4181;

  unsigned int fib[20];
  fib[0] = 0;
  fib[1] = 1;
  for (int i = 2; i < 20; i++) {
    fib[i] = fib[i - 1] + fib[i - 2];
  }

  for (int i = 0; i < 20; i++) {
    if (fib[i] != expected[i]) {
      puts("FAIL at index ");
      puthex((unsigned int)i);
      puts(": got 0x");
      puthex(fib[i]);
      puts(" expected 0x");
      puthex(expected[i]);
      putchar('\n');
```

```
      pass = 0;
    }
  }

  puts("Printing sequence for demo output (fib(1..5) and fib(19)):\n");
  for (int i = 1; i <= 5; i++) {
    puts("fib(");
    putdec((unsigned int)i);
    puts(") = ");
    putdec(fib[i]);
    puts(" (0x");
    puthex(fib[i]);
    putchar(')');
    putchar('\n');
  }

  puts("fib(19) = ");
  putdec(fib[19]);
  puts(" (0x");
  puthex(fib[19]);
  putchar(')');
  putchar('\n');

  if (pass) {
    puts("PASS: All Fibonacci numbers correct!\n");
  } else {
    puts("FAIL: Fibonacci sequence errors\n");
  }

  sim_halt();
  return 0;
}
```

B.0.5 fibonacci_test.c

```
// Fibonacci test - computes first 20 Fibonacci numbers
#include "simple_system_common.h"

int main(void) {
    puts("=== Fibonacci Test ===\n");
    int pass = 1;

    // Expected first 20 Fibonacci numbers (use volatile to avoid memcpy)
    volatile unsigned int expected[20];
    expected[0]=0; expected[1]=1; expected[2]=1; expected[3]=2; expected[4]=3;
    expected[5]=5; expected[6]=8; expected[7]=13; expected[8]=21; expected[9]=34;
    expected[10]=55; expected[11]=89; expected[12]=144; expected[13]=233; expected[14]=377;
    expected[15]=610; expected[16]=987; expected[17]=1597; expected[18]=2584; expected[19]=4181;

    unsigned int fib[20];
    fib[0] = 0;
    fib[1] = 1;
    for (int i = 2; i < 20; i++) {
        fib[i] = fib[i-1] + fib[i-2];
    }

    for (int i = 0; i < 20; i++) {
        if (fib[i] != expected[i]) {
            puts("FAIL at index ");
            puthex(i);
            puts(": got 0x");
            puthex(fib[i]);
            puts(" expected 0x");
            puthex(expected[i]);
            putchar('\n');
            pass = 0;
```

```
        }
    }

    puts("fib(19) = 0x");
    puthex(fib[19]);
    putchar('\n');

    if (pass) {
        puts("PASS: All Fibonacci numbers correct!\n");
    } else {
        puts("FAIL: Fibonacci sequence errors\n");
    }

    sim_halt();
    return 0;
}
```

B.0.6 fibonacci_verbose_test.c

```
// Fibonacci verbose demo test - prints fib(1) through fib(10), and fib(19)
#include "simple_system_common.h"

static void putdec(unsigned int value) {
    char buf[10];
    int pos = 0;

    if (value == 0) {
        putchar('0');
        return;
    }

    while (value > 0 && pos < (int)sizeof(buf)) {
        buf[pos++] = (char)('0' + (value % 10));
        value /= 10;
    }

    while (pos > 0) {
        putchar(buf[--pos]);
    }
}

int main(void) {
    puts("=== Fibonacci Verbose Demo Test ===\n");
    int pass = 1;

    volatile unsigned int expected[20];
    expected[0] = 0;    expected[1] = 1;    expected[2] = 1;    expected[3] = 2;    expected[4] = 3;
    expected[5] = 5;    expected[6] = 8;    expected[7] = 13;   expected[8] = 21;   expected[9] = 34;
    expected[10] = 55;  expected[11] = 89;  expected[12] = 144; expected[13] = 233; expected[14] = 377;
    expected[15] = 610; expected[16] = 987; expected[17] = 1597; expected[18] = 2584; expected[19] = 4181;

    unsigned int fib[20];
    fib[0] = 0;
    fib[1] = 1;
    for (int i = 2; i < 20; i++) {
        fib[i] = fib[i - 1] + fib[i - 2];
    }

    for (int i = 0; i < 20; i++) {
        if (fib[i] != expected[i]) {
            puts("FAIL at index ");
            puthex((unsigned int)i);
            puts(": got 0x");
            puthex(fib[i]);
            puts(" expected 0x");
            puthex(expected[i]);
```

```
            putchar('\n');
            pass = 0;
        }
    }

    puts("Printing sequence for demo output (fib(1..10) and fib(19)):\n");
    for (int i = 1; i <= 10; i++) {
        puts("fib(");
        putdec((unsigned int)i);
        puts(") = ");
        putdec(fib[i]);
        puts(" (0x");
        puthex(fib[i]);
        putchar(')');
        putchar('\n');
    }

    puts("fib(19) = ");
    putdec(fib[19]);
    puts(" (0x");
    puthex(fib[19]);
    putchar(')');
    putchar('\n');

    if (pass) {
        puts("PASS: All Fibonacci numbers correct!\n");
    } else {
        puts("FAIL: Fibonacci sequence errors\n");
    }

    sim_halt();
    return 0;
}
```

B.0.7 gls_branch_trap_pipe_stress_test.c

```
#include "simple_system_common.h"

volatile uint32_t g_trap_count = 0;
volatile uint32_t g_trap_ecall = 0;
volatile uint32_t g_trap_illegal = 0;
volatile uint32_t g_trap_misc = 0;
volatile uint32_t g_sink = 0;

static void trap_handler(void) __attribute__((naked, aligned(4)));

static void trap_handler(void) {
  asm volatile(
      "csrr t0, mcause\n"
      "la   t1, g_trap_count\n"
      "lw   t2, 0(t1)\n"
      "addi t2, t2, 1\n"
      "sw   t2, 0(t1)\n"

      "li   t3, 11\n"  // ecall from M-mode
      "beq  t0, t3, 1f\n"
      "li   t3, 2\n"   // illegal instruction
      "beq  t0, t3, 2f\n"
      "j    3f\n"

      "1:\n"
      "la   t1, g_trap_ecall\n"
      "lw   t2, 0(t1)\n"
      "addi t2, t2, 1\n"
      "sw   t2, 0(t1)\n"
      "j    4f\n"
```

```
      "2:\n"
      "la   t1, g_trap_illegal\n"
      "lw   t2, 0(t1)\n"
      "addi t2, t2, 1\n"
      "sw   t2, 0(t1)\n"
      "j    4f\n"

      "3:\n"
      "la   t1, g_trap_misc\n"
      "lw   t2, 0(t1)\n"
      "addi t2, t2, 1\n"
      "sw   t2, 0(t1)\n"

      "4:\n"
      // Skip faulting instruction and continue.
      "csrr t0, mepc\n"
      "addi t0, t0, 4\n"
      "csrw mepc, t0\n"
      "mret\n");
}

static inline void install_trap_handler(void) {
  asm volatile("csrw mtvec, %0" : : "r"(trap_handler));
}

static uint32_t branch_mix(uint32_t seed) {
  uint32_t x = seed;
  for (uint32_t i = 0; i < 256; ++i) {
    if ((i & 1u) == 0u) {
      x ^= (i << 3);
    } else {
      x += (i * 7u);
    }

    if ((x & 0x3u) == 0u) {
      x = (x << 1) | (x >> 31);
    } else if ((x & 0x3u) == 1u) {
      x = (x >> 1) | (x << 31);
    } else if ((x & 0x3u) == 2u) {
      x ^= 0xA5A55A5Au;
    } else {
      x += 0x13579BDFu;
    }
  }
  return x;
}

int main(void) {
  puts("=== GLS Branch+Trap Pipeline Stress Test ===\n");

  install_trap_handler();

  volatile uint32_t *ram_w = (volatile uint32_t *)0x00100400;
  volatile uint16_t *ram_h = (volatile uint16_t *)0x00100400;
  volatile uint8_t *ram_b = (volatile uint8_t *)0x00100400;
  volatile uint32_t *sim_out = (volatile uint32_t *)0x00020000;

  uint32_t v = 0x12345678u;

  for (uint32_t outer = 0; outer < 64; ++outer) {
    v = branch_mix(v + outer);

    // Mixed LSU traffic to keep pipeline/LSU interaction busy.
    ram_w[0] = v;
    ram_h[(outer + 1u) & 1u] = (uint16_t)(v >> 8);
    ram_b[(outer + 2u) & 3u] = (uint8_t)(v >> 16);
    g_sink ^= ram_w[0] ^ ram_h[outer & 1u] ^ ram_b[outer & 3u];
```

```
    // Branchy output traffic.
    if (outer & 1u) {
      *sim_out = (uint32_t)('A' + (outer & 15u));
    } else {
      *sim_out = (uint32_t)('a' + (outer & 15u));
    }

    // Frequent trap entry/exit with resume.
    asm volatile("ecall");
    asm volatile(".word 0x00000000");  // intentionally illegal
  }

  // Keep counters in memory for waveform/log inspection if needed.
  g_sink ^= g_trap_count ^ g_trap_ecall ^ g_trap_illegal ^ g_trap_misc;

  puts("PASS: Branch+trap stress complete\n");
  sim_halt();
  return 0;
}
```

B.0.8 gls_mmio_be_cov_followup_test.c

```
// GLS MMIO + byte-enable closure test
// Combines strong MMIO stimulus with a controlled unmapped sweep to close
// remaining cp_data_be/cp_mmio_reg/x_be_region bins.
#include "simple_system_common.h"

static void fault_skip_handler(void) __attribute__((naked, aligned(4)));

static void fault_skip_handler(void) {
  asm volatile(
      "csrr t0, mepc\n"
      "addi t0, t0, 4\n"
      "csrw mepc, t0\n"
      "mret\n");
}

static inline void install_fault_handler(void) {
  asm volatile("csrw mtvec, %0" : : "r"(fault_skip_handler));
}

static inline void fault_load8(uintptr_t addr) {
  asm volatile("lb zero, 0(%0)" : : "r"(addr) : "memory");
}

static inline void fault_load16(uintptr_t addr) {
  asm volatile("lh zero, 0(%0)" : : "r"(addr) : "memory");
}

static inline void fault_load32(uintptr_t addr) {
  asm volatile("lw zero, 0(%0)" : : "r"(addr) : "memory");
}

static inline void fault_store8(uintptr_t addr, uint8_t value) {
  asm volatile("sb %1, 0(%0)" : : "r"(addr), "r"(value) : "memory");
}

static inline void fault_store16(uintptr_t addr, uint16_t value) {
  asm volatile("sh %1, 0(%0)" : : "r"(addr), "r"(value) : "memory");
}

static inline void fault_store32(uintptr_t addr, uint32_t value) {
  asm volatile("sw %1, 0(%0)" : : "r"(addr), "r"(value) : "memory");
}
```

```
static void be_write_sweep(volatile uint8_t *base) {
  base[0] = 0x11;
  base[1] = 0x22;
  base[2] = 0x33;
  base[3] = 0x44;

  *(volatile uint16_t *)(base + 0) = 0xA1B2;
  *(volatile uint16_t *)(base + 2) = 0xC3D4;

  *(volatile uint32_t *)(base + 0) = 0x55667788;
}

static inline void touch_mmio_rw_word(volatile uint32_t *p, uint32_t w) {
  (void)*p;
  *p = w;
}

int main(void) {
  puts("=== GLS MMIO/BE Closure Test ===\n");

  install_fault_handler();

  volatile uint8_t *ram_b = (volatile uint8_t *)0x00100180;
  volatile uint32_t *ram_w = (volatile uint32_t *)0x00100180;

  volatile uint8_t *sim_out_b = (volatile uint8_t *)0x00020000;
  volatile uint32_t *sim_out_w = (volatile uint32_t *)0x00020000;
  volatile uint32_t *sim_ctrl_w = (volatile uint32_t *)0x00020008;
  volatile uint32_t *sim_other_w = (volatile uint32_t *)0x00020004;

  volatile uint8_t *tm_cmp_lo_b = (volatile uint8_t *)0x00030008;
  volatile uint32_t *tm_mtime_lo_w = (volatile uint32_t *)0x00030000;
  volatile uint32_t *tm_mtime_hi_w = (volatile uint32_t *)0x00030004;
  volatile uint32_t *tm_cmp_lo_w = (volatile uint32_t *)0x00030008;
  volatile uint32_t *tm_cmp_hi_w = (volatile uint32_t *)0x0003000C;
  volatile uint32_t *tm_other_w = (volatile uint32_t *)0x00030010;

  // 1) Full byte-enable sweep in RAM + MMIO regions.
  be_write_sweep(ram_b);
  be_write_sweep(sim_out_b);
  be_write_sweep(tm_cmp_lo_b);

  // 2) Force all mmio_reg bins with read+write pairs.
  (void)*ram_w;
  *ram_w ^= 0xFFFFFFFFu;
  touch_mmio_rw_word(sim_out_w, 'F');
  (void)*sim_ctrl_w;
  *sim_ctrl_w = 0;
  touch_mmio_rw_word(sim_other_w, 0x13579BDFu);
  touch_mmio_rw_word(tm_mtime_lo_w, 0x00001000u);
  touch_mmio_rw_word(tm_mtime_hi_w, 0x00000000u);
  touch_mmio_rw_word(tm_cmp_lo_w, 0x00002000u);
  touch_mmio_rw_word(tm_cmp_hi_w, 0x00000000u);
  touch_mmio_rw_word(tm_other_w, 0x2468ACE0u);

  // 3) Probe unmapped region with all store BE patterns to close x_be_region.
  // Handler skips each faulting instruction and keeps test alive.
  const uintptr_t hole = 0x40000000u;

  fault_store8(hole + 0, 0x11u);
  fault_store8(hole + 1, 0x22u);
  fault_store8(hole + 2, 0x33u);
  fault_store8(hole + 3, 0x44u);
  fault_store16(hole + 0, 0xA1B2u);
  fault_store16(hole + 2, 0xC3D4u);
  fault_store32(hole + 0, 0x55667788u);
  fault_load32(hole + 0);
```

```
  // 4) Extra handshakes to diversify crosses.
  for (int i = 0; i < 32; ++i) {
    *ram_b = (uint8_t)(*ram_b + (uint8_t)i);
    *sim_out_w = (uint32_t)('a' + (i & 15));
    (void)*tm_mtime_lo_w;
  }

  puts("PASS: MMIO/BE closure sequence complete\n");
  sim_halt();
  return 0;
}
```

B.0.9 gls_mmio_be_cov_strong_test.c

```
// Strong GLS MMIO + byte-enable coverage test
// Targets cg_core_lite_mmio_be with broad, systematic stimulus.
#include "simple_system_common.h"

static void be_write_sweep(volatile uint8_t *base) {
  // byte0..byte3
  base[0] = 0x11;
  base[1] = 0x22;
  base[2] = 0x33;
  base[3] = 0x44;

  // half0, half1
  *(volatile uint16_t *)(base + 0) = 0xA1B2;
  *(volatile uint16_t *)(base + 2) = 0xC3D4;

  // word
  *(volatile uint32_t *)(base + 0) = 0x55667788;
}

static void touch_mmio_rw_word(volatile uint32_t *p, uint32_t w) {
  (void)*p; // read
  *p = w;   // write
}

int main(void) {
  puts("=== GLS MMIO/BE Strong Coverage Test ===\n");

  // Non-MMIO (RAM) region
  volatile uint8_t *ram_b = (volatile uint8_t *)0x00100100;
  volatile uint32_t *ram_w = (volatile uint32_t *)0x00100100;

  // SimCtrl region
  volatile uint8_t *sim_out_b = (volatile uint8_t *)0x00020000;   // simctrl_out
  volatile uint32_t *sim_out_w = (volatile uint32_t *)0x00020000;
  volatile uint32_t *sim_ctrl_w = (volatile uint32_t *)0x00020008; // simctrl_ctrl
  volatile uint32_t *sim_other_w = (volatile uint32_t *)0x00020004; // mmio_other

  // Timer region
  volatile uint8_t *tm_cmp_lo_b = (volatile uint8_t *)0x00030008;
  volatile uint32_t *tm_mtime_lo_w = (volatile uint32_t *)0x00030000;
  volatile uint32_t *tm_mtime_hi_w = (volatile uint32_t *)0x00030004;
  volatile uint32_t *tm_cmp_lo_w = (volatile uint32_t *)0x00030008;
  volatile uint32_t *tm_cmp_hi_w = (volatile uint32_t *)0x0003000C;
  volatile uint32_t *tm_other_w = (volatile uint32_t *)0x00030010; // mmio_other

  // 1) Sweep all BE patterns in RAM, SimCtrl, Timer.
  be_write_sweep(ram_b);
  be_write_sweep(sim_out_b);
  be_write_sweep(tm_cmp_lo_b);

  // 2) Ensure non-mmio read+write path is exercised.
  (void)*ram_w;
```

```
  *ram_w = *ram_w ^ 0xFFFFFFFFu;

  // 3) Exercise each MMIO sub-register bin with read+write.
  // simctrl_out
  touch_mmio_rw_word(sim_out_w, 'S');

  // simctrl_ctrl: write zero only (avoid accidental halt)
  (void)*sim_ctrl_w;
  *sim_ctrl_w = 0;

  // mmio_other in simctrl page
  touch_mmio_rw_word(sim_other_w, 0x13579BDFu);

  // timer mtime/cmp registers
  touch_mmio_rw_word(tm_mtime_lo_w, 0x00001000u);
  touch_mmio_rw_word(tm_mtime_hi_w, 0x00000000u);
  touch_mmio_rw_word(tm_cmp_lo_w, 0x00002000u);
  touch_mmio_rw_word(tm_cmp_hi_w, 0x00000000u);

  // mmio_other in timer page
  touch_mmio_rw_word(tm_other_w, 0x2468ACE0u);

  // 4) Extra handshakes to diversify cross coverage.
  for (int i = 0; i < 32; ++i) {
    ram_b[i & 3] = (uint8_t)(i + 1);
    (void)*tm_mtime_lo_w;
    *sim_out_w = (uint32_t)('A' + (i & 15));
  }

  puts("PASS: MMIO/BE strong coverage sequence complete\n");
  sim_halt();
  return 0;
}
```

B.0.10 gls_mmio_be_cov_test.c

```
// GLS MMIO + byte-enable coverage test
// Targets cg_core_lite_mmio_be bins in ibex_core_wrapper_netlist.sv
#include "simple_system_common.h"

int main(void) {
  puts("=== GLS MMIO/BE Coverage Test ===\n");

  volatile uint32_t *ram_w = (volatile uint32_t *)0x00100080;
  volatile uint16_t *ram_h0 = (volatile uint16_t *)0x00100080;
  volatile uint16_t *ram_h1 = (volatile uint16_t *)0x00100082;
  volatile uint8_t *ram_b0 = (volatile uint8_t *)0x00100080;
  volatile uint8_t *ram_b1 = (volatile uint8_t *)0x00100081;
  volatile uint8_t *ram_b2 = (volatile uint8_t *)0x00100082;
  volatile uint8_t *ram_b3 = (volatile uint8_t *)0x00100083;

  volatile uint32_t *sim_out_w = (volatile uint32_t *)0x00020000;
  volatile uint16_t *sim_out_h = (volatile uint16_t *)0x00020000;
  volatile uint8_t *sim_out_b = (volatile uint8_t *)0x00020000;
  volatile uint32_t *sim_ctrl_w = (volatile uint32_t *)0x00020008;

  volatile uint32_t *tm_mtime_lo_w = (volatile uint32_t *)0x00030000;
  volatile uint32_t *tm_mtime_hi_w = (volatile uint32_t *)0x00030004;
  volatile uint32_t *tm_cmp_lo_w = (volatile uint32_t *)0x00030008;
  volatile uint32_t *tm_cmp_hi_w = (volatile uint32_t *)0x0003000C;
  volatile uint16_t *tm_cmp_lo_h = (volatile uint16_t *)0x00030008;
  volatile uint8_t *tm_cmp_lo_b = (volatile uint8_t *)0x00030008;

  // RAM: drive byte/halfword/word write enables.
  *ram_w = 0x11223344u;
  *ram_h0 = 0xA1B2u;
```

```
  *ram_h1 = 0xC3D4u;
  *ram_b0 = 0x11u;
  *ram_b1 = 0x22u;
  *ram_b2 = 0x33u;
  *ram_b3 = 0x44u;

  // RAM reads to keep read side active.
  (void)*ram_w;
  (void)*ram_h0;
  (void)*ram_b2;

  // SimCtrl: output register using multiple access widths.
  *sim_out_w = 'W';
  *sim_out_h = 'H';
  *sim_out_b = 'B';
  (void)*sim_out_w;

  // SimCtrl control register: keep at zero to avoid halting.
  *sim_ctrl_w = 0;
  (void)*sim_ctrl_w;

  // Timer MMIO: hit each key register and multiple widths.
  uint32_t tlo = *tm_mtime_lo_w;
  uint32_t thi = *tm_mtime_hi_w;
  *tm_cmp_hi_w = thi;
  *tm_cmp_lo_w = tlo + 2000u;
  *tm_cmp_lo_h = 0x55AAu;
  *tm_cmp_lo_b = 0x77u;

  (void)*tm_mtime_lo_w;
  (void)*tm_mtime_hi_w;
  (void)*tm_cmp_lo_w;
  (void)*tm_cmp_hi_w;

  // A short loop to generate extra read/write handshakes.
  for (int i = 0; i < 16; ++i) {
    *ram_w = *ram_w + (uint32_t)i;
    *tm_cmp_lo_w = *tm_cmp_lo_w + (uint32_t)i;
    (void)*tm_mtime_lo_w;
  }

  puts("PASS: MMIO/BE coverage sequence complete\n");
  sim_halt();
  return 0;
}
```

B.0.11 gls_pipe_flow_cov_test.c

```
// GLS pipeline-flow stress test
// Targets stage-flow and pipe-flow covergroup crosses with varied control/data flow.
#include "simple_system_common.h"

static volatile uint32_t sink;

typedef uint32_t (*fn_t)(uint32_t);

static uint32_t add3(uint32_t x) { return x + 3u; }
static uint32_t mul5(uint32_t x) { return x * 5u; }

static uint32_t branch_mix(uint32_t x) {
  uint32_t acc = x;
  for (uint32_t i = 0; i < 64; ++i) {
    if ((i & 1u) == 0u) {
      acc ^= (i << 1);
    } else {
      acc += (i * 3u);
```

```
    }

    if ((acc & 7u) == 3u) {
      acc = (acc << 1) | (acc >> 31);
    } else if ((acc & 7u) == 5u) {
      acc = (acc >> 1) | (acc << 31);
    }
  }
  return acc;
}

int main(void) {
  puts("=== GLS Pipeline Flow Coverage Test ===\n");

  volatile uint8_t *ram_b = (volatile uint8_t *)0x00100200;
  volatile uint16_t *ram_h = (volatile uint16_t *)0x00100200;
  volatile uint32_t *ram_w = (volatile uint32_t *)0x00100200;
  volatile uint32_t *sim_out = (volatile uint32_t *)0x00020000;

  // Mixed-width load/store traffic to keep LSU and WB active.
  for (uint32_t i = 0; i < 128; ++i) {
    ram_b[(i + 0) & 3u] = (uint8_t)(i + 0x11u);
    ram_h[(i + 1) & 1u] = (uint16_t)(0x1200u + i);
    ram_w[0] = 0xA5A50000u ^ i;

    sink ^= ram_b[(i + 2) & 3u];
    sink ^= ram_h[(i + 0) & 1u];
    sink ^= ram_w[0];

    if ((i & 15u) == 0u) {
      *sim_out = (uint32_t)('P' + (i & 7u));
    }
  }

  // Direct + indirect calls encourage jal/jalr usage.
  fn_t f = add3;
  uint32_t v = 1u;
  for (uint32_t i = 0; i < 64; ++i) {
    f = (i & 1u) ? mul5 : add3;
    v = f(v);
    v = branch_mix(v);
  }

  // A small loop with explicit dependency chains keeps pipeline transitions active.
  for (uint32_t i = 0; i < 128; ++i) {
    v ^= (v << 5) + (v >> 2) + i;
    if (v & 1u) {
      v += 0x1020304u;
    } else {
      v ^= 0x00FF00FFu;
    }
    sink = v;
  }

  puts("PASS: pipeline-flow coverage sequence complete\n");
  sim_halt();
  return 0;
}
```

B.0.12 gls_quick_cov_test.c

```
// Quick GLS coverage sanity test
// Focuses on realistic coverpoints after irq-vector bins were disabled.
#include "simple_system_common.h"

int main(void) {
```

```
  puts("=== GLS Quick Coverage Test ===\n");

  volatile uint32_t *ram32 = (volatile uint32_t *)0x00100040;
  volatile uint16_t *ram16 = (volatile uint16_t *)0x00100041; // addr[1:0]=01
  volatile uint8_t  *ram8  = (volatile uint8_t  *)0x00100043; // addr[1:0]=11

  volatile uint32_t *sim_out  = (volatile uint32_t *)0x00020000;
  volatile uint32_t *sim_ctrl = (volatile uint32_t *)0x00020008;

  volatile uint32_t *tm_mtime_lo  = (volatile uint32_t *)0x00030000;
  volatile uint32_t *tm_mtime_hi  = (volatile uint32_t *)0x00030004;
  volatile uint32_t *tm_cmp_lo    = (volatile uint32_t *)0x00030008;
  volatile uint32_t *tm_cmp_hi    = (volatile uint32_t *)0x0003000C;

  // RAM read/write with multiple low address bit patterns.
  uint32_t v = *ram32;
  *ram32 = v ^ 0x55AA55AAu;
  *ram16 = 0xBEEFu;
  *ram8  = 0x5Au;

  // SimCtrl read/write (write 0 to control to avoid halting).
  (void)*sim_out;
  *sim_out = 'Q';
  *sim_ctrl = 0;

  // Timer read/write.
  uint32_t tlo = *tm_mtime_lo;
  uint32_t thi = *tm_mtime_hi;
  *tm_cmp_hi = thi;
  *tm_cmp_lo = tlo + 1000u;

  // A few extra memory handshakes.
  for (int i = 0; i < 8; ++i) {
    *ram32 = *ram32 + (uint32_t)i;
    (void)*tm_mtime_lo;
  }

  puts("PASS: quick coverage sequence complete\n");
  sim_halt();
  return 0;
}
```

B.0.13 irq_misaligned_test.c

```
// ASCII output demo for ibex_simple_system
#include "simple_system_common.h"

int main(void) {
  puts("=== Ibex ASCII Demo (Simple System) ===\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%@@#*+======++#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%#+-------=*#@@@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*--------+@@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%**%=-------=%%%%%%%-%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%#==#=--------#%%%%%%*---+%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*--**--------=%%%%%%@#+---=#@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%*---*+--------+%%%%%%%%%%%-%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%#=---**--------+%%%%%%%%%%%%#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%#----=%---------#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%#-----**--------=%%%%%%%%%%%%%%%%%%%%####%%%%%=*%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%#------#=--------=%%%%%%%%%%%%%%%+=----------+=-+%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%@+------%=---------#%%%%%%%%%%%#=-----------------%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%#=------#+---------=%%%%%%%%%%=-------------------*@%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%###-------*#----------+%%%%%%@*---------------------=%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%+**-------=%+----------=%%%%%=-----------------------*%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%=+*--------+#------------#%%---------------##*=------+%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%#-=#=--------*%------------=%#-------------%%%%%*-----=%%*+%%%%%%%%%%%%%%%%%%%%%%\n");
```

```
  puts("%%%%%%%%%%%%%%%%%%%%*--#+---------+#-------------+%*-----------@%%%%%+---=#%=---*%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%*--=#=---------=%+-------------*%+---------=%%%%%++#%+------+@%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%*---+*-----------+%=-------------+%+--------=%@%%@*=--------+@%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%#----*#------------*#=-------------+%+---------==-----------*%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%=----+#=------------*%+-------------=%*----------------+---#%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%*------##-------------=#%+------------=%*--------------+*---#%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%=------=##--------------+%#+----------+#-------------##---%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%@*---------+#*=--------------=*%#+--------#+----------*@#---%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%*-----------+%%=----------------*@*------**--------=%%#=-=%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%#-------------=+#%#+--------------*@+----#=------=%%#*--+%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%*----------------=+*%%#*+=---------+%=--=*-----=%%*----+%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%#*@#=----------------------+%@%+-----=#--=#----+%%%=---=%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%=-*%#=----------------------=%%=---*+-=#---*%%%##%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*---=*%#*+=------------------%*--**-%+-=%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%+------=*#%@%##*+++====-----=@+-#+**=#@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%+-------------------=+#%=--@+*#*#*@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*=-------------------#+*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*=----------------##%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%@@#+=---------=+@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%@@%%%@@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%@@@@@%%%%%%%%@@@@%%%%%%%%%%%%%@@@@@@@@@%%%%%%%%@@@@@@@@@@@@@@@@%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%#-...-%%%%%%%%*:...+%%%%%%%%%*-:..   ..:=#@%%%%%-...............+@%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%#:   :%%%%%%%%*.   +%%%%%%%=.             .*%%%%:              .+@%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%#:   :%%%%%%%%*.   +%%%%%#.    -+***+-.  .+@%%%%:   :===========#%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%#:   :%%%%%%%%*.   +%%%%#.    =%%%%%%%%#=#%%%%%%:   :%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%#:   :%%%%%%%%*.   +%%%@-.   =@%%%%%%%%%%%%%%%%%:   :#%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%#:   :%%%%%%%%*.   +%%%@:   .#%%%%%%%%%%%%%%%%%%:             :#%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%#:   :%%%%%%%%*.   +%%%@:   .#%%%%%%%%%%%%%%%%%%:             :#%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%-   .%%%%%%%%+    +%%%@=.   -@%%%%%%%%%%%%%%%%%:   :#%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%@=.   -%%%%%@%.  .*@%%%%.    .%%%%%%%@*.+@%%%%%:   :%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%#:    .:--:.     =@%%%%%@:     .:---:    .*%%%%:   :%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%=.           .*%%%%%%%%@*:            .=%%%%%:   :%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%@#-.......=%@%%%%%%%%%%%%@+:.......=@@%%%%%%-...-%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");

  puts("\nPASS: ASCII demo completed\n");
  sim_halt();
  return 0;
}
```

B.0.14 minimal_timer_test.c

```
// Timer specific test - minimal, just tries 0x30000
#include "simple_system_common.h"

int main(void) {
    puts("=== Minimal Timer Test ===\n");

    // First: read from RAM to confirm baseline
    volatile uint32_t *ram_ptr = (volatile uint32_t *)0x100200;
    uint32_t rv = *ram_ptr;
    puts("RAM ok\n");

    // Second: read from SimCtrl (works)
    volatile uint32_t *sc = (volatile uint32_t *)0x20000;
    uint32_t sv = *sc;
    puts("SimCtrl ok\n");

    // Third: try 0x30000 (timer MTIME)
    puts("Trying 0x30000...\n");
    volatile uint32_t *t = (volatile uint32_t *)0x30000;
    uint32_t tv = *t;
```

```
    puthex(tv);
    putchar('\n');

    puts("PASS!\n");
    sim_halt();
    return 0;
}
```

B.0.15 NGDemo.c

```
// ASCII output demo for ibex_simple_system (NG logo)
#include "simple_system_common.h"

int main(void) {
  puts("=== Welcome to NG RAD: Next-Generation RISC-V ASIC Design’s Senior Design Showcase!===\n");
  puts("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@%###########################**#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@+...........................-#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@+...........................-#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@+...........................-#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@+:::::::::::::::::::::......-#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@%=......-#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@%=......-#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@%=......-#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@%#######@@@@@@@@@@@@@@#######%@@@@@@@@@@@%*+--:::--=+#@@@@@@@@@@@@@@%=......-#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@=.......-#@@@@@@@@@@@#.......*@@@@@@@@@%+:.... .........-#@@@@@@@@@@@%=......-#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@=.........+@@@@@@@@@@#.......*@@@@@@%=...................:#@@@@@@@@@@%=......-#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@=..........-%@@@@@@@#.......*@@@@@*......................-%@@@@@@@%=......-#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@=.....    ....*@@@@@@#.......*@@@@*........=%@@%%@@*:......+%@@@@@@@%=......-#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@=.............=%@@@@#.......*@@@*:......:%@@@@@@@@@@%*:-*@@@@@@@@@@@@%=......-#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@=..............:#@@@#.......*@@%-......:#@@@@@@@@@@@@@@@@@@@@@@@@@@@@%=......-#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@=......::........+@@#.......*@@#:......-%@@@@@#++++++++++++++%@@@@@%=......-#@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@=......:#*........:%#.......*@@#:......=@@@@@@+.............:*@@@@@%+:::::=%@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@=......:#@%-........=.......*@@#:......=%@@@@@=.............:#@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@=......:#@@%+..............*@@%-......:%@@@@@+::::::.......-%@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@=......:#@@@@#:........    .*@@@+.......=@@@@@@@@@@@@#:......=%@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@=......:#@@@@@%=.......    .*@@@%=.......-#%@@@@@@%*:......-%@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@=......:#@@@@@@@+:.....    .*@@@@@=.........-=**=-........:#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@=......:#@@@@@@@@#-....  ...*@@@@@@*:....................-#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@=......:#@@@@@@@@@%+........*@@@@@@@%*-................-#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@#+++++++*%@@@@@@@@@@@#+++++++%@@@@@@@@@@%*=-:.....::-+*%@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@%%%@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");
  puts("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");


  puts("\nPASS: NG ASCII demo completed\n");
  sim_halt();
  return 0;
}
```

B.0.16 periph_read_test.c

```
// Peripheral read test - tries reading from MMIO regions
// Tests whether any non-RAM load works through the netlist
#include "simple_system_common.h"

int main(void) {
    puts("=== Periph Read Test ===\n");

    // First, read from RAM (should always work)
    volatile uint32_t *ram_ptr = (volatile uint32_t *)0x100200;
    uint32_t ram_val = *ram_ptr;
    puts("RAM read ok: 0x");
```

```
    puthex(ram_val);
    putchar('\n');

    // Now try reading from a KNOWN location in peripheral space
    // SimCtrl is at 0x20000 - it's write-only but let's try a read
    volatile uint32_t *simctrl_ptr = (volatile uint32_t *)0x20000;
    puts("Attempting SimCtrl read (0x20000)...\n");
    uint32_t sc_val = *simctrl_ptr;
    puts("SimCtrl read: 0x");
    puthex(sc_val);
    putchar('\n');

    puts("Both reads done!\n");
    sim_halt();
    return 0;
}
```

B.0.17 riscv_instr_test.c

```
// Instruction test - exercises ALU, memory, branches, MUL/DIV
#include "simple_system_common.h"

// Prevent compiler from optimizing away
volatile int sink;

int main(void) {
    puts("=== RISC-V Instruction Test ===\n");
    int pass = 1;
    int a, b, r;

    // --- ALU ops ---
    a = 100; b = 42;
    r = a + b;
    if (r != 142) { puts("FAIL: ADD\n"); pass = 0; }

    r = a - b;
    if (r != 58) { puts("FAIL: SUB\n"); pass = 0; }

    r = a & b;
    if (r != (100 & 42)) { puts("FAIL: AND\n"); pass = 0; }

    r = a | b;
    if (r != (100 | 42)) { puts("FAIL: OR\n"); pass = 0; }

    r = a ^ b;
    if (r != (100 ^ 42)) { puts("FAIL: XOR\n"); pass = 0; }

    r = a << 3;
    if (r != 800) { puts("FAIL: SLL\n"); pass = 0; }

    r = a >> 2;
    if (r != 25) { puts("FAIL: SRL\n"); pass = 0; }

    // Signed right shift
    a = -100;
    r = a >> 2;
    if (r != -25) { puts("FAIL: SRA\n"); pass = 0; }

    // SLT
    a = -5; b = 5;
    r = (a < b) ? 1 : 0;
    if (r != 1) { puts("FAIL: SLT\n"); pass = 0; }

    // SLTU
    r = ((unsigned int)a < (unsigned int)b) ? 1 : 0;
    if (r != 0) { puts("FAIL: SLTU\n"); pass = 0; } // -5 as unsigned is huge
```

```
if (pass) puts("ALU: PASS\n"); else puts("ALU: FAIL\n");

// --- Memory ops ---
pass = 1;
volatile int mem_word = 0xDEADBEEF;
if (mem_word != (int)0xDEADBEEF) { puts("FAIL: SW/LW\n"); pass = 0; }

volatile short mem_half = (short)0x1234;
if (mem_half != 0x1234) { puts("FAIL: SH/LH\n"); pass = 0; }

volatile char mem_byte = (char)0x42;
if (mem_byte != 0x42) { puts("FAIL: SB/LB\n"); pass = 0; }

// Unsigned load half
volatile unsigned short mem_uhal = 0xF234;
if (mem_uhal != 0xF234) { puts("FAIL: LHU\n"); pass = 0; }

// Unsigned load byte
volatile unsigned char mem_ubyt = 0xF2;
if (mem_ubyt != 0xF2) { puts("FAIL: LBU\n"); pass = 0; }

if (pass) puts("Memory: PASS\n"); else puts("Memory: FAIL\n");

// --- Branches ---
pass = 1;
a = 10; b = 20;
if (!(a == 10)) { puts("FAIL: BEQ\n"); pass = 0; }
if (!(a != b))  { puts("FAIL: BNE\n"); pass = 0; }
if (!(a < b))   { puts("FAIL: BLT\n"); pass = 0; }
if (!(b >= a))  { puts("FAIL: BGE\n"); pass = 0; }

if (pass) puts("Branch: PASS\n"); else puts("Branch: FAIL\n");

// --- MUL/DIV (M extension) ---
pass = 1;
a = 123; b = 456;
r = a * b;
if (r != 56088) { puts("FAIL: MUL\n"); pass = 0; }

// Signed multiply negative
a = -7; b = 13;
r = a * b;
if (r != -91) { puts("FAIL: MUL neg\n"); pass = 0; }

a = 1000; b = 7;
r = a / b;
if (r != 142) { puts("FAIL: DIV\n"); pass = 0; }

r = a % b;
if (r != 6) { puts("FAIL: REM\n"); pass = 0; }

// Signed division
a = -100; b = 7;
r = a / b;
if (r != -14) { puts("FAIL: DIV neg\n"); pass = 0; }

r = a % b;
if (r != -2) { puts("FAIL: REM neg\n"); pass = 0; }

// Division by zero (RISC-V: returns -1 for div, dividend for rem)
a = 42; b = 0;
// Use asm to prevent compiler from optimizing
asm volatile("div %0, %1, %2" : "=r"(r) : "r"(a), "r"(b));
if (r != -1) { puts("FAIL: DIV by zero\n"); pass = 0; }
asm volatile("rem %0, %1, %2" : "=r"(r) : "r"(a), "r"(b));
if (r != 42) { puts("FAIL: REM by zero\n"); pass = 0; }
```

```
    if (pass) puts("MUL/DIV: PASS\n"); else puts("MUL/DIV: FAIL\n");

    // --- AUIPC/LUI ---
    pass = 1;
    unsigned int pc_val;
    asm volatile("auipc %0, 0" : "=r"(pc_val));
    // PC should be in the range 0x00100000+ (RAM region)
    if (pc_val < 0x00100000 || pc_val > 0x00200000) {
        puts("FAIL: AUIPC out of range\n"); pass = 0;
    }

    unsigned int lui_val;
    asm volatile("lui %0, 0xDEADB" : "=r"(lui_val));
    if (lui_val != 0xDEADB000) { puts("FAIL: LUI\n"); pass = 0; }

    if (pass) puts("AUIPC/LUI: PASS\n"); else puts("AUIPC/LUI: FAIL\n");

    puts("\n=== All instruction tests complete ===\n");
    sim_halt();
    return 0;
}
```

B.0.18 sieve_test.c

```
// Sieve of Eratosthenes test - find primes up to 200
#include "simple_system_common.h"

#define LIMIT 200

// Use volatile to prevent compiler from using memset/memcpy
volatile char is_prime[LIMIT + 1];

int main(void) {
    puts("=== Sieve of Eratosthenes Test ===\n");

    // Initialize manually (avoid memset)
    for (int i = 0; i <= LIMIT; i++) {
        is_prime[i] = 1;
    }
    is_prime[0] = 0;
    is_prime[1] = 0;

    // Sieve
    for (int i = 2; i * i <= LIMIT; i++) {
        if (is_prime[i]) {
            for (int j = i * i; j <= LIMIT; j += i) {
                is_prime[j] = 0;
            }
        }
    }

    // Count primes and verify
    int count = 0;
    for (int i = 2; i <= LIMIT; i++) {
        if (is_prime[i]) {
            count++;
        }
    }

    puts("Number of primes up to 200: ");
    puthex(count);
    putchar('\n');

    // There are exactly 46 primes up to 200
    if (count == 46) {
        puts("PASS: Correct prime count (46)!\n");
```

```
    } else {
        puts("FAIL: Expected 46 primes\n");
    }

    // Verify a few known primes manually
    int pass = 1;
    if (!is_prime[2])   { puts("FAIL: 2 should be prime\n"); pass = 0; }
    if (!is_prime[3])   { puts("FAIL: 3 should be prime\n"); pass = 0; }
    if (!is_prime[5])   { puts("FAIL: 5 should be prime\n"); pass = 0; }
    if (!is_prime[7])   { puts("FAIL: 7 should be prime\n"); pass = 0; }
    if (!is_prime[11])  { puts("FAIL: 11 should be prime\n"); pass = 0; }
    if (!is_prime[13])  { puts("FAIL: 13 should be prime\n"); pass = 0; }
    if (!is_prime[97])  { puts("FAIL: 97 should be prime\n"); pass = 0; }
    if (!is_prime[101]) { puts("FAIL: 101 should be prime\n"); pass = 0; }
    if (!is_prime[197]) { puts("FAIL: 197 should be prime\n"); pass = 0; }
    if (!is_prime[199]) { puts("FAIL: 199 should be prime\n"); pass = 0; }

    // Verify a few known non-primes
    if (is_prime[4])   { puts("FAIL: 4 should not be prime\n"); pass = 0; }
    if (is_prime[6])   { puts("FAIL: 6 should not be prime\n"); pass = 0; }
    if (is_prime[100]) { puts("FAIL: 100 should not be prime\n"); pass = 0; }
    if (is_prime[150]) { puts("FAIL: 150 should not be prime\n"); pass = 0; }
    if (is_prime[200]) { puts("FAIL: 200 should not be prime\n"); pass = 0; }

    if (pass) {
        puts("PASS: All prime checks correct!\n");
    } else {
        puts("FAIL: Prime check errors\n");
    }

    sim_halt();
    return 0;
}
```

B.0.19 timer_irq_mmio_test.c

```
// Timer IRQ + MMIO coverage test
// Keep timer interrupts live while exercising RAM, Timer, and SimCtrl accesses.
#include "simple_system_common.h"

static inline void fence_io(void) {
    asm volatile("fence iorw, iorw" : : : "memory");
}

int main(void) {
    puts("=== Timer IRQ + MMIO Coverage Test ===\n");

    // Keep the timer interrupt pending, but do not enable machine interrupts.
    // The coverage model only needs the raw irq signal asserted.
    timer_disable();
    uint64_t trigger_time = timer_read() + 5000;
    timecmp_update(trigger_time);

    volatile uint32_t *ram_base = (volatile uint32_t *)0x00100000;
    volatile uint32_t *simctrl_out = (volatile uint32_t *)0x20000;
    volatile uint32_t *simctrl_ctrl = (volatile uint32_t *)0x20008;
    volatile uint32_t *timer_lo = (volatile uint32_t *)0x30000;
    volatile uint32_t *timer_hi = (volatile uint32_t *)0x30004;
    volatile uint32_t *timer_cmp_lo = (volatile uint32_t *)0x30008;
    volatile uint32_t *timer_cmp_hi = (volatile uint32_t *)0x3000C;

    // Keep touching memory while waiting for the timer compare threshold to
    // pass.
    fence_io();

    for (int i = 0; i < 4096; ++i) {
```

```
        uint32_t offset = (uint32_t)(i & 7);
        volatile uint32_t *word_ptr = (volatile uint32_t *)((uintptr_t)ram_base + (offset * 4));
        volatile uint16_t *half_ptr = (volatile uint16_t *)((uintptr_t)ram_base + (offset * 2) + 1);
        volatile uint8_t *byte_ptr = (volatile uint8_t *)((uintptr_t)ram_base + offset + 3);

        uint32_t word = *word_ptr;
        *word_ptr = word ^ (0x01010101u * (uint32_t)(i + 1));
        *half_ptr = (uint16_t)(0xA500u ^ (uint16_t)i);
        *byte_ptr = (uint8_t)(0x5Au ^ (uint8_t)i);

        // Touch both timer and simctrl regions while the timer IRQ is armed.
        (void)*timer_lo;
        (void)*timer_hi;
        *simctrl_out = (uint32_t)('A' + (i & 15));

        if ((i & 15) == 0) {
            *simctrl_ctrl = 0;
        }

        if (timer_read() >= trigger_time) {
            break;
        }
    }

    // Stop the timer interrupt source before the final MMIO and alignment sweep.
    timer_disable();

    for (int i = 0; i < 32; ++i) {
        (void)*timer_lo;
        (void)*timer_hi;
        *ram_base = *ram_base + (uint32_t)i;
        *timer_cmp_hi = 0x00000000;
        *timer_cmp_lo = 0x00002000 + (uint32_t)i;
    }

    puts("PASS: Timer IRQ + MMIO coverage test completed!\n");
    sim_halt();
    return 0;
}
```

B.0.20 timer_test.c

```
// Timer read test - just reads the timer without enabling interrupts
#include "simple_system_common.h"

int main(void) {
    puts("=== Timer Read Test ===\n");

    // Try a simple read from timer MTIME (0x30000)
    puts("Reading MTIME...\n");
    volatile uint32_t *timer_lo = (volatile uint32_t *)0x30000;
    volatile uint32_t *timer_hi = (volatile uint32_t *)0x30004;

    uint32_t tlo = *timer_lo;
    puts("MTIME_LO: 0x");
    puthex(tlo);
    putchar('\n');

    uint32_t thi = *timer_hi;
    puts("MTIME_HI: 0x");
    puthex(thi);
    putchar('\n');

    // Read again to see if it's incrementing
    tlo = *timer_lo;
    puts("MTIME_LO (2nd): 0x");
```

```
    puthex(tlo);
    putchar('\n');

    // Now try a write to MTIMECMP
    puts("Writing MTIMECMP...\n");
    volatile uint32_t *cmp_lo = (volatile uint32_t *)0x30008;
    volatile uint32_t *cmp_hi = (volatile uint32_t *)0x3000C;
    *cmp_hi = 0xFFFFFFFF;
    *cmp_lo = 0xFFFFFFFF;
    puts("MTIMECMP set to max\n");

    // Read back
    uint32_t clo = *cmp_lo;
    puts("MTIMECMP_LO: 0x");
    puthex(clo);
    putchar('\n');

    puts("PASS: Timer access works!\n");
    sim_halt();
    return 0;
}
```

B.0.21 UCFDemo.c

```
// ASCII output demo for ibex_simple_system
#include "simple_system_common.h"

int main(void) {
  puts("=== Welcome to NG RAD: Next-Generation RISC-V ASIC Design's Senior Design Showcase!===\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%@@#*+======++#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%#+-------=*#@@@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*--------+@@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%**%=-------=%%%%%%-%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%#==#=--------#%%%%%%*---+%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*--**--------=%%%%%@#+---=#@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%*---*+--------+%%%%%%%%%%%-%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%#=---**--------+%%%%%%%%%%%%#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%#----=%---------#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%#-----**---------=%%%%%%%%%%%%%%%%%%%####%%%%%=*%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%#------#=--------=%%%%%%%%%%%%%%%%%+=----------+=-+%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%@+------%=---------#%%%%%%%%%%%%#=----------------%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%#=------#+---------=%%%%%%%%%%=-------------------*@%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%###-------*#----------+%%%%%%@*---------------------=%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%+**--------=%+---------=%%%%%=-----------------------*%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%=+*---------+#-----------#%%---------------##*=------+%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%#-=#=--------*%------------=%#-------------%%%%%*-----=%%*+%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%*--#+---------+#-------------+%*-----------@%%%%%+---=#%=---*%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%*--=#=---------=%+-------------*%+---------=%%%%%%++#%+------+@%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%*---+*-----------+%=-------------+%+--------=%@%%%@*=--------+@%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%#----*#------------*#=-------------+%+---------==----------*%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%=----+#=------------*%+-------------=%*-----------------+---#%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%*------##-------------=#%+------------=%*--------------+*---#%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%=------=##=--------------+%#+----------+#-------------##---%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%@*---------+#*=-------------=*%#+--------#+----------*@#---%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%*-----------+%%=---------------*@*-------**--------=%%#=-=%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%#-------------=+#%#+--------------*@+----#=------=%%#*--+%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%*----------------=+*%%#*+=---------+%=--=*-----=%%*----+%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%#*@#=-----------------+%@%+-----=#--=#----+%%%=---=%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%=-*%#=------------------=%%=---*+-=#---*%%%##%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%*---=*%#*+=-------------------%*--**-%+-=%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%+------=*#%@%##*+++====-----=@+-#+**=#@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%+-------------------=+#%=--@+*#*#*@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*=-------------------#+*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*=-----------------##%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%@@#+=--------=+@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%@@%%%@@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
```

```
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%@@@@@%%%%%%%%%@@@@%%%%%%%%%%%%@@@@@@@@%%%%%%%%%@@@@@@@@@@@@@@@@%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%#-...-%%%%%%%%*:...+%%%%%%%%%*-:..   ..:=#@%%%%%-...............+@%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%#:   :%%%%%%%%*.   +%%%%%%%=.             .*%%%%:              .+@%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%#:   :%%%%%%%%*.   +%%%%%#.     -+**+-.  .+@%%%%:   :===========#%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%#:   :%%%%%%%%*.   +%%%%#.    =%%%%%%%%#=#%%%%%%:   :%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%#:   :%%%%%%%%*.   +%%%@-.   =@%%%%%%%%%%%%%%%%%:   :#%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%#:   :%%%%%%%%*.   +%%%@:   .#%%%%%%%%%%%%%%%%%%:             :#%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%#:   :%%%%%%%%*.   +%%%@:   .#%%%%%%%%%%%%%%%%%%:             :#%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%-   .%%%%%%%%+    +%%%@=.   -@%%%%%%%%%%%%%%%%%:   :#%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%@=.   -%%%%%@%.   .*@%%%%.    .%%%%%%%@*.+@%%%%%:   :%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%#:    .:--:.     =@%%%%%@:     .:---:    .*%%%%:   :%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%=.           .*%%%%%%%%@*:            .=%%%%%:   :%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
  puts("%%%%%%%%%%%%%%%%%%%%%%@#-.......=%@%%%%%%%%%%%%@+:.......=@@%%%%%%-...-%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");

  puts("\nPASS: ASCII demo completed\n");
  sim_halt();
  return 0;
}
```

Chapter C - ChatGPT Prompts and Outcomes

C.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.

C.2 Case Study 2: Ibex DV Setup Chat Transcript

This appendix contains an excerpted transcript of an interactive ChatGPT session used to set up the Ibex UVM/DV co-simulation flow on the VLSI server. The conversation is shown as a sequence of user prompts and assistant responses.

Exchange 1

User:

I have a VLSI server with access to cadence tools. From scratch, I'd like to run Ibex's DV flow, as described on this page `https://github.com/lowRISC/ibex/tree/master/dv/uvm/core_ibex` Ideally, I would have a python venv set up for this. Assume that I have none of the dependencies installed or set up yet, besides the VLSI server with cadence tools.

Assistant:

Here's a from-scratch "Ibex DV on the VLSI server" recipe, tuned to your setup (Cadence tools + Python venv + Ibex DV/UVM co-sim).

I'll assume:

- Ibex root: `/home/net/al663069/ws/ibex`
- You want a venv inside there: `/home/net/al663069/ws/ibex/.venv`
- Cadence Xcelium (`xrun`) is already installed via your VLSI modules
- You're using `bash` or something similar

You can adapt paths if you like, but try to stay consistent.

0. One-time sanity checks

On the VLSI server:

```
# Check Python
python3 --version

# Check Xcelium
xrun -version
```

You need Python 3 and working `xrun`. Ibex's docs only require "a SystemVerilog simulator that supports UVM" and say VCS is tested, but Xcelium/Incisive work fine via the `SIMULATOR` switch.

1. Clone Ibex

```
mkdir -p /home/net/al663069/ws
cd /home/net/al663069/ws

git clone https://github.com/lowRISC/ibex.git
cd ibex
export IBEX_ROOT="$PWD"
```

(If you already have the repo cloned, just set `IBEX_ROOT` accordingly.)

2. Create & populate the Python venv

Ibex provides a `python-requirements.txt` that pulls in the version of FuseSoC and other tools it expects. Docs explicitly say to install via that file, not a separate FuseSoC install.

```
cd "$IBEX_ROOT"

python3 -m venv .venv
source .venv/bin/activate

# Always good practice
pip install -U pip wheel

# Install Ibex / FuseSoC / helper tools
pip install -U -r python-requirements.txt
```

After this, you should have `fusesoc`, etc., in `.venv/bin`.

You don't need to manually set `PYTHONPATH` for these tools just to run Ibex DV — the Makefiles set up script paths for you. (I'll come back to your `PYTHONPATH` question at the end.)

3. Install a RISC-V GCC toolchain

Ibex DV uses RISCV-DV to generate programs, and you need a RISC-V GCC to compile them. You can either use a pre-built toolchain or build the RISCV GNU toolchain and set environment variables `RISCV_TOOLCHAIN`, `RISCV_GCC`, and `RISCV_OBJCOPY`.

C.2.0.0.1 Option A (recommended): Pre-built toolchain

1. Make a tools directory:

   ```
   mkdir -p $HOME/tools
   cd $HOME/tools
   ```

2. Download an appropriate RV32 pre-built toolchain tarball (e.g., from lowRISC's toolchain releases or a standard riscv-gnu-toolchain build).

3. Unpack it somewhere under `$HOME/tools`, e.g.:

   ```
   tar -xf riscv32-toolchain.tar.xz
   mv riscv32-unknown-elf-toolchain riscv32-toolchain
   ```

4. Set environment variables:

   ```
   export RISCV_TOOLCHAIN="$HOME/tools/riscv32-toolchain"
   export PATH="$RISCV_TOOLCHAIN/bin:$PATH"

   export RISCV_GCC="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-gcc"
   export RISCV_OBJCOPY="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-objcopy"
   ```

   The DV flow will use these variables when building tests.

4. Build the lowRISC Spike fork for co-simulation

Ibex DV's co-sim requires a specific Spike fork and build configuration:

- Repo: lowRISC-specific `riscv-isa-sim` fork
- Version: branch/tag named `ibex_cosim`
- Configure flags: `--enable-commitlog --enable-misaligned`

C.2.0.0.2 4.1 Clone & checkout correct Spike

```
cd /home/net/al663069/ws

git clone https://github.com/lowRISC/riscv-isa-sim.git spike-ibex
cd spike-ibex

# Use the ibex-compatible version mentioned in the docs
git checkout ibex_cosim
```

(If `git checkout ibex_cosim` fails, use `git tag` / `git branch -a` to find the exact name they use in the current repo.)

C.2.0.0.3 4.2 Configure & build

```
mkdir build
cd build

# Install into $HOME/tools/spike-ibex
../configure \
  --prefix=$HOME/tools/spike-ibex \
  --enable-commitlog \
  --enable-misaligned

make -j$(nproc)
make install
```

That installs Spike + its libraries + pkg-config files into `$HOME/tools/spike-ibex`.

C.2.0.0.4 4.3 Set `SPIKE_PATH` and `PKG_CONFIG_PATH`

```
export SPIKE_PATH="$HOME/tools/spike-ibex/bin"
export PKG_CONFIG_PATH="$HOME/tools/spike-ibex/lib/pkgconfig
${PKG_CONFIG_PATH:+:$PKG_CONFIG_PATH}"
```

Quick sanity check:

```
pkg-config --exists riscv-disasm && echo "riscv-disasm OK" || echo
"riscv-disasm NOT FOUND"
pkg-config --exists riscv-fesvr   && echo "riscv-fesvr OK"   || echo
"riscv-fesvr NOT FOUND"
```

5. (Optional) Create an `env-ibex.sh` helper

You don't want to re-export all this by hand forever. In your home directory:

```
cat > $HOME/env-ibex.sh <<'EOF'
# Ibex DV + tools environment

export IBEX_ROOT=/home/net/al663069/ws/ibex

# Activate Python venv
if [ -f "$IBEX_ROOT/.venv/bin/activate" ]; then
  # Only source if not already active
  case "$VIRTUAL_ENV" in
    "$IBEX_ROOT/.venv") ;;
    *) . "$IBEX_ROOT/.venv/bin/activate" ;;
  esac
fi

# RISC-V toolchain
export RISCV_TOOLCHAIN="$HOME/tools/riscv32-toolchain"
export PATH="$RISCV_TOOLCHAIN/bin:$PATH"
export RISCV_GCC="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-gcc"
export RISCV_OBJCOPY="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-objcopy"

# Spike for Ibex
export SPIKE_PATH="$HOME/tools/spike-ibex/bin"
export PKG_CONFIG_PATH="$HOME/tools/spike-ibex/lib/pkgconfig$
{PKG_CONFIG_PATH:+:$PKG_CONFIG_PATH}"

# (Optional) Add Ibex DV scripts to PYTHONPATH { not strictly required
export PYTHONPATH="$IBEX_ROOT/dv/uvm/core_ibex:
$IBEX_ROOT/vendor/google_riscv-dv:${PYTHONPATH:-}"

EOF
```

Then in any new shell:

```
source $HOME/env-ibex.sh
```

6. Point Ibex DV at Xcelium instead of VCS

The core DV docs show you can pick any supported RTL simulator by editing `dv/uvm/core_ibex/yaml/simulator.yaml` and using `make ... SIMULATOR=xxx`.

C.2.0.0.5 6.1 Find the simulator key for Xcelium

```
cd "$IBEX_ROOT/dv/uvm/core_ibex"

# Just inspect the YAML locally
grep -n "xrun" yaml/simulator.yaml || grep -n "ius" yaml/simulator.yaml
```

Look for an entry like:

```
- name: xrun
  compile_cmd: xrun ...
  sim_cmd:     xrun ...
```

Whatever the `name:` is (e.g. `xrun`, `ius`, or `xcelium`), that's what you pass as `SIMULATOR=`.

On many Xcelium setups, a line like this in the YAML works:

```
compile_cmd: xrun -64bit -uvm -access +rwc <cmp_opts> ...
sim_cmd:     xrun -64bit -R -uvm <sim_opts> \
-svseed <seed> +UVM_TESTNAME=<rtl_test> ...
```

You'll tune this based on how your VLSI team invokes Xcelium.

7. Run Ibex DV / co-simulation

At this point you should have:

- `xrun` in your `PATH`
- `RISCV_*` env vars pointing at a working RV32 toolchain
- `SPIKE_PATH` and `PKG_CONFIG_PATH` pointing at the lowRISC Spike install
- Python venv active with Ibex's python requirements installed

Now:

```
cd "$IBEX_ROOT/dv/uvm/core_ibex"

# Optional: check the sim driver help
python3 sim.py --help
```

C.2.0.0.6 7.1 Run a small single test first Replace `<cadence_key>` with the simulator name (e.g. `xlm`, `xrun`, etc.):

```
# single random test, one iteration, co-sim enabled
make SIMULATOR=<cadence_key> \
     TEST=riscv_machine_mode_rand_test \
     ITERATIONS=1 \
     SEED=1
```

You can add:

- `WAVES=1` to dump waves
- `OUT=out`$_x$*celium_testtoredirectoutputdirectory*
- `VERBOSE=1 for more logging`

Example:

```
make SIMULATOR=<cadence_key> \
     TEST=riscv_machine_mode_rand_test \
     ITERATIONS=1 \
     SEED=123 \
     WAVES=1 \
     OUT=out_xcelium_smoke
```

If everything is wired correctly, you'll see the flow:

1. Build RISCV-DV generator
2. Generate assembly
3. Compile with your RISC-V GCC
4. Compile testbench + DUT
5. Run Xcelium RTL + Spike in lockstep and compare traces

Once that works, a full regression is just:

```
make SIMULATOR=<cadence_key>
```

8. Original `PYTHONPATH` question

For the Ibex DV Makefile flow, `PYTHONPATH` can actually be empty; the Makefiles and scripts work fine with just the current working directory and their relative imports.

If you want to be explicit, something like this is reasonable:

```
export IBEX_ROOT=/home/net/al663069/ws/ibex

export PYTHONPATH="$IBEX_ROOT/dv/uvm/core_ibex:
$IBEX_ROOT/vendor/google_riscv-dv:${PYTHONPATH:-}"
```

If your venv is at `/home/net/al663069/ibex/ibex-venv`, just source it first:

```
source /home/net/al663069/ibex/ibex-venv/bin/activate
export IBEX_ROOT=/home/net/al663069/ws/ibex
export PYTHONPATH="$IBEX_ROOT/dv/uvm/core_ibex:
$IBEX_ROOT/vendor/google_riscv-dv:${PYTHONPATH:-}"
```

Exchange 2

User:

`https://github.com/lowRISC/lowrisc-toolchains/releases/tag/20220210-1` this is the toolchain i wish to use

Assistant:

That release is exactly what the Ibex docs describe for using a pre-built toolchain.

On the 20220210-1 release page there are multiple toolchains (rv32imc, rv32imcb, rv64imac, etc.). For Ibex DV you want a 32-bit, bare-metal GCC toolchain, for example the Linux/x86_64 *rv32imc gcc* tarball.

Assuming you downloaded it to:

```
/home/net/al663069/tools/lowrisc-toolchain-gcc-rv32imc-20220210-1.tar.xz
```

Install it into `$HOME/.local`:

```
mkdir -p ~/.local

tar -xJf /home/net/al663069/tools/
lowrisc-toolchain-gcc-rv32imc-20220210-1.tar.xz \
    --strip-components=1 -C ~/.local
```

After this, you should have:

```
~/.local/bin/riscv32-unknown-elf-gcc
~/.local/bin/riscv32-unknown-elf-objcopy
...
```

Set the environment variables:

```
export RISCV_TOOLCHAIN=$HOME/.local
export RISCV_GCC="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-gcc"
export RISCV_OBJCOPY="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-objcopy"
export PATH="$RISCV_TOOLCHAIN/bin:$PATH"
```

You can optionally wrap this into `ibex_env.sh`:

```
cat << 'EOF' > /home/net/al663069/ws/ibex/ibex_env.sh
export IBEX_ROOT=/home/net/al663069/ws/ibex

# lowRISC pre-built toolchain (20220210-1 rv32imc)
export RISCV_TOOLCHAIN=$HOME/.local
export PATH="$RISCV_TOOLCHAIN/bin:$PATH"
export RISCV_GCC="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-gcc"
export RISCV_OBJCOPY="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-objcopy"

# (You'll add SPIKE_PATH + PKG_CONFIG_PATH later once Spike is built)
EOF
```

Then do:

```
source /home/net/al663069/ws/ibex/ibex_env.sh
```

Sanity checks:

```
which riscv32-unknown-elf-gcc
riscv32-unknown-elf-gcc --version
echo "$RISCV_GCC"
echo "$RISCV_OBJCOPY"
```

This toolchain is independent of your Python venv (which only handles Python packages).

Exchange 3

User:

everything seems to be working in this sanity check, and i am in my python venv. what would be next? it looks like you mentioned walking through SPIKE from the lowRISC fork+ the exact env youi'd need for the co-sim and DV flow line up

Assistant:

Great, that means the toolchain + venv side is in good shape. Next is to build lowRISC's Spike fork, wire it into the environment, and run a small Ibex DV co-sim test (with Xcelium).

Assumptions:

- Ibex root: `/home/net/al663069/ws/ibex`
- Workspace: `/home/net/al663069/ws`
- You're already in your Python venv

A. Build the lowRISC Spike fork (`ibex_cosim`)

From any shell:

```
cd /home/net/al663069/ws

# Get the co-sim Spike
git clone -b ibex_cosim https://github.com/lowRISC/riscv-isa-sim.git \
riscv-isa-sim-cosim

cd riscv-isa-sim-cosim
mkdir build
cd build

# Configure and build Spike for Ibex co-sim
../configure \
  --enable-commitlog \
  --enable-misaligned \
```

```
  --prefix=$HOME/tools/spike-ibex

make -j$(nproc)
make install
```

B. Wire Spike into your environment

In the same shell (with venv active):

```
export SPIKE_PATH="$HOME/tools/spike-ibex/bin"
export PKG_CONFIG_PATH="$HOME/tools/spike-ibex/lib/pkgconfig$
{PKG_CONFIG_PATH:+:$PKG_CONFIG_PATH}"
```

Sanity checks:

```
which spike
"$SPIKE_PATH/spike" --help   # spike --version may not be supported

pkg-config --exists riscv-disasm && echo "riscv-disasm OK" || echo
"riscv-disasm NOT FOUND"
pkg-config --exists riscv-fesvr   && echo "riscv-fesvr OK"   || echo
"riscv-fesvr NOT FOUND"
```

C. Add this to your reusable Ibex env

Extend `ibex_env.sh`:

```
cat > /home/net/al663069/ws/ibex/ibex_env.sh << 'EOF'
# Ibex DV environment

export IBEX_ROOT=/home/net/al663069/ws/ibex

# RISC-V toolchain (lowRISC prebuilt you already installed)
export RISCV_TOOLCHAIN=$HOME/.local
export PATH="$RISCV_TOOLCHAIN/bin:$PATH"
export RISCV_GCC="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-gcc"
export RISCV_OBJCOPY="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-objcopy"

# Spike (lowRISC fork, ibex_cosim branch)
export SPIKE_PATH="$HOME/tools/spike-ibex/bin"
export PKG_CONFIG_PATH="$HOME/tools/spike-ibex/lib/pkgconfig$
{PKG_CONFIG_PATH:+:$PKG_CONFIG_PATH}"

# (Optional) Python package search path for Ibex DV + RISCV-DV
export PYTHONPATH="$IBEX_ROOT/dv/uvm/core_ibex:
$IBEX_ROOT/vendor/google_riscv-dv:${PYTHONPATH:-}"
EOF
```

Then for an Ibex DV session:

```
# 1) Activate Python venv
source /home/net/al663069/ibex/ibex-venv/bin/activate

# 2) Load Ibex env
source /home/net/al663069/ws/ibex/ibex_env.sh
```

D. Run a small Ibex DV + co-sim test with Xcelium

From dv/uvm/core_ibex:

```
cd /home/net/al663069/ws/ibex/dv/uvm/core_ibex

make SIMULATOR=xlm \
     TEST=riscv_machine_mode_rand_test \
     ITERATIONS=1 \
     COSIM=1 \
     SEED=1
```

With waves:

```
make SIMULATOR=xlm \
     TEST=riscv_machine_mode_rand_test \
     ITERATIONS=1 \
     COSIM=1 \
     SEED=1 \
     WAVES=1 \
     OUT=out_xlm_cosim_smoke
```

If that fails, you would inspect the error logs; if it passes, you have a working co-sim setup.

Exchange 4

User:

```
(ibex-venv) [al663069@vlsi build] export SPIKE_PATH=
"$HOME/tools/spike-ibex/bin"
(ibex-venv) [al663069@vlsi build] export PKG_CONFIG_PATH=
"$HOME/tools/spike-ibex/lib/pkgconfig${PKG_CONFIG_PATH:+:$PKG_CONFIG_PATH}"
(ibex-venv) [al663069@vlsi build] which spike
/usr/bin/which: no spike in (/home/net/al663069
/.local/bin:/home/net/al663069/.local/bin:
/home/net/al663069/ibex/ibex-venv/bin:/home/net/al663069/ibex/riscv-gnu/bin:
/home/net/al663069/.local/bin:/home/net/al663069/bin:...
:/home/net/xilinx/Vivado/2019.1/bin)
(ibex-venv) [al663069@vlsi build] "$SPIKE_PATH/spike" --version
/home/net/al663069/tools/spike-ibex/bin/spike: unrecognized option --version
```

```
Try 'spike --help' for more information.
(ibex-venv) [al663069@vlsi build]
(ibex-venv) [al663069@vlsi build] pkg-config --exists riscv-disasm &&
echo "riscv-disasm OK" || echo "riscv-disasm NOT FOUND"
riscv-disasm OK
(ibex-venv) [al663069@vlsi build] pkg-config --exists riscv-fesvr   &&
echo "riscv-fesvr OK"   || echo "riscv-fesvr NOT FOUND"
riscv-fesvr OK
(ibex-venv) [al663069@vlsi build] pwd
/home/net/al663069/ws/riscv-isa-sim-cosim/build
(ibex-venv) [al663069@vlsi build]
```

Assistant:

This looks good; the outputs just need interpretation.

- `which spike` fails because `SPIKE_PATH` is not on `PATH`. Ibex DV doesn't need that, but if you want:

  ```
  export PATH="$SPIKE_PATH:$PATH"
  which spike
   $SPIKE\_PATH/spike" --version fails because this Spike build
   doesn't support --version}. That's fine; use:
  "$SPIKE_PATH/spike" --help
  ```

- The important part is:

  ```
  riscv-disasm OK
  riscv-fesvr OK
  ```

  indicating `pkg-config` can see the Spike libraries.

At this point you can run an Ibex DV co-sim test with Xcelium (assuming toolchain and env vars are set):

```
cd /home/net/al663069/ws/ibex/dv/uvm/core_ibex

make SIMULATOR=xlm \
     TEST=riscv_machine_mode_rand_test \
     ITERATIONS=1 \
     COSIM=1 \
     SEED=1
```

Optionally:

```
make SIMULATOR=xlm \
     TEST=riscv_machine_mode_rand_test \
     ITERATIONS=1 \
     COSIM=1 \
     SEED=1 \
     WAVES=1 \
     OUT=out_xlm_cosim_smoke
```

Exchange 5

User:

```
cat > /home/net/al663069/ws/ibex/ibex_env.sh << 'EOF'
# Ibex DV environment

export IBEX_ROOT=/home/net/al663069/ws/ibex

# RISC-V toolchain (lowRISC prebuilt you already installed)
export RISCV_TOOLCHAIN=$HOME/.local
export PATH="$RISCV_TOOLCHAIN/bin:$PATH"
export RISCV_GCC="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-gcc"
export RISCV_OBJCOPY="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-objcopy"

# Spike (lowRISC fork, ibex_cosim branch)
export SPIKE_PATH="$HOME/tools/spike-ibex/bin"
export PKG_CONFIG_PATH="$HOME/tools/spike-ibex/lib/pkgconfig
${PKG_CONFIG_PATH:+:$PKG_CONFIG_PATH}"

# (Optional) Python package search path for Ibex DV + RISCV-DV
export PYTHONPATH="$IBEX_ROOT/dv/uvm/core_ibex:
$IBEX_ROOT/vendor/google_riscv-dv:${PYTHONPATH:-}"
EOF

should i also add spike to path in this?
```

Assistant:

You don't need to add Spike to `PATH` for Ibex DV, but it's convenient. I'd modify the script to:

```
cat > /home/net/al663069/ws/ibex/ibex_env.sh << 'EOF'
# Ibex DV environment

export IBEX_ROOT=/home/net/al663069/ws/ibex

# RISC-V toolchain (lowRISC prebuilt you already installed)
export RISCV_TOOLCHAIN=$HOME/.local
export PATH="$RISCV_TOOLCHAIN/bin:$PATH"

export RISCV_GCC="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-gcc"
export RISCV_OBJCOPY="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-objcopy"

# Spike (lowRISC fork, ibex_cosim branch)
export SPIKE_PATH="$HOME/tools/spike-ibex/bin"
export PKG_CONFIG_PATH="$HOME/tools/spike-ibex/lib/pkgconfig
${PKG_CONFIG_PATH:+:$PKG_CONFIG_PATH}"
```

```
# Also put Spike on PATH for convenience
export PATH="$SPIKE_PATH:$PATH"

# (Optional) Python package search path for Ibex DV + RISCV-DV
export PYTHONPATH="$IBEX_ROOT/dv/uvm/core_ibex:
$IBEX_ROOT/vendor/google_riscv-dv:${PYTHONPATH:-}"
EOF
```

Then:

```
source /home/net/al663069/ibex/ibex-venv/bin/activate
source /home/net/al663069/ws/ibex/ibex_env.sh

which spike
which riscv32-unknown-elf-gcc
```

Exchange 6

User:

all 3 sample tests you gave me ran properly and passed, thank you so much. where can I see all of the available tests/the code that makes them up?

Assistant:

There are two main places:

C.2.0.0.7 1. The full list of available tests

From `dv/uvm/core_ibex`:

```
cd $IBEX_ROOT/dv/uvm/core_ibex

less riscv_dv_extension/testlist.yaml
grep '^- test:' riscv_dv_extension/testlist.yaml
```

Every `TEST=...` you can use with `make` comes from the `test:` fields in this YAML. You can list just test names:

```
grep '^- test:' riscv_dv_extension/testlist.yaml | awk '{print $3}'
```

C.2.0.0.8 2. The SystemVerilog code that implements the tests

The UVM testbench code is in:

```
cd $IBEX_ROOT/dv/uvm/core_ibex
ls tests
```

Key files:

- `tests/core_ibex_base_test.sv`: base class for all tests.

- `tests/core_ibex_test_lib.sv`: concrete UVM tests derived from the base class.
- `tests/core_ibex_seq_lib.sv`: sequence library (interrupt and debug sequences, etc.).

To trace one test end-to-end, inspect its entry in `riscv_dv_extension/testlist.yaml`, note the `uvm_test` name, and then search that class name in the `tests/` directory.

RISCV-DV base classes and configs live under `vendor/google_riscv-dv`:

```
cd $IBEX_ROOT/vendor/google_riscv-dv
ls src
ls yaml
```

Ibex's `riscv_dv_extension/testlist.yaml` maps RISCV-DV tests to Ibex-specific UVM tests.

Exchange 7

User:

where can i see the waveform dump of any test i run?

Assistant:

By default, everything (logs and waves) lands under the output directory, e.g.:

```
dv/uvm/core_ibex/out/
```

If you specify `OUT=out_xlm_cosim_smoke`, you'll see:

```
dv/uvm/core_ibex/out_xlm_cosim_smoke/
  seed-1/
    rtl_sim/
      riscv_machine_mode_rand_test.0/
        sim.log
        <waveform database>
```

Make sure you actually enabled waves with `WAVES=1`:

```
cd $IBEX_ROOT/dv/uvm/core_ibex

make SIMULATOR=xlm \
     TEST=riscv_machine_mode_rand_test \
     ITERATIONS=1 \
     SEED=1 \
     COSIM=1 \
     WAVES=1 \
     OUT=out_xlm_cosim_smoke
```

Then inspect:

```
cd out_xlm_cosim_smoke/seed-1/rtl_sim/riscv_machine_mode_rand_test.0
ls
```

On Xcelium, you'll typically see an SHM database (e.g., `xcelium.shm` or `waves.shm`) containing the waveform data.

Exchange 8

User:

in the directory `/home/net/al663069/ws/ibex/dv/uvm/core_ibex/out_xlm_cosim_smoke/run/tests/riscv_machine_mode_rand_test.123/waves.shm/` i have two files. i'd like to use xrun's GUI if possible. the two files are `waves.dsn` and `waves.trn`

Assistant:

That's exactly what you should see for an Xcelium SHM database:

```
.../waves.shm/
  waves.dsn
  waves.trn
```

- `waves.shm` is the SHM directory.
- `waves.dsn` holds design/signal info.
- `waves.trn` holds the waveform transitions.

Xcelium's GUI for waves is SimVision, which `xrun -gui` launches under the hood. Since the simulation has already finished, you can open the SHM database directly in SimVision.

From the test directory one level above `waves.shm`:

```
cd /home/net/al663069/ws/ibex/dv/uvm/core_ibex/ \
  out_xlm_cosim_smoke/run/tests/riscv_machine_mode_rand_test.123

simvision -waves waves.shm &
```

Alternatively:

```
simvision waves.shm/waves.trn &
# or
simvision waves.shm/waves.dsn &
```

Once SimVision opens:

1. Open a waveform window (if one is not already open).
2. Use the design browser to find the Ibex DUT and relevant signals.
3. Drag and drop signals into the waveform window.

You are now viewing the simulation results using the Xcelium/SimVision GUI.

C.3 Case Study 3: Generating RTL-file hierarchy order for Synthesis Scripts

Exchange 1

User:

Can you order these files and output them into a .txt file such that designs that are lower in the design hierarchy are at the top and top modules and .sv files that use module instantiations are at the bottom. The reason the order is important is because we are using this order to input these files into a synthesis tool that cares about the order.

(Workspace structure summarized; key RTL files include `ibex_core.sv`, `ibex_top.sv`, `ibex_controller.sv`, etc.)

Assistant: High-level response: infer RTL dependencies and generate a bottom-up, synthesis-friendly file order for the Ibex core, then write it to a text file in the `rtl` directory.

Actions taken (conceptual):

- Located Ibex RTL files in `rtl/`.
- Inspected key modules (`ibex_core.sv`, `ibex_top.sv`, `ibex_controller.sv`) and existing filelist `ibex_core.f`.
- Derived a bottom-up ordering: leaf/utility modules at the top, `ibex_core.sv` / `ibex_top.sv` at the bottom.
- Created a synthesis-order text file in `rtl/`.

Resulting file created:

- `rtl/ibex_synth_order.txt`

Its contents (simplified, one file per line in bottom-up order):

```
// Ibex RTL files in bottom-up synthesis order
// Leaf / utility modules first, top-level last

// Primitive / package
ibex_pkg.sv
ibex_csr.sv
ibex_counter.sv

// Register files
ibex_register_file_ff.sv
ibex_register_file_fpga.sv
ibex_register_file_latch.sv

// ALU / mult/div
ibex_alu.sv
ibex_multdiv_slow.sv
ibex_multdiv_fast.sv
```

```
// Fetch / prefetch / icache
ibex_fetch_fifo.sv
ibex_prefetch_buffer.sv
ibex_icache.sv
ibex_branch_predict.sv

// LSU and PMP
ibex_pmp.sv
ibex_load_store_unit.sv

// CSRs and misc
ibex_csr.sv
ibex_cs_registers.sv
ibex_dummy_instr.sv
ibex_tracer_pkg.sv
ibex_tracer.sv

// Pipeline stages and controller
ibex_decoder.sv
ibex_compressed_decoder.sv
ibex_ex_block.sv
ibex_wb_stage.sv
ibex_id_stage.sv
ibex_if_stage.sv
ibex_controller.sv

// Core and top-levels
ibex_core.sv
ibex_lockstep.sv
ibex_top_tracing.sv
ibex_top.sv
```

User 2 I want this for the ibex simple system using the small configuration. Exclude files that would not synthesize because of the given parameters.

(Additional context: `baseline_ibex/examples/simple_system/rtl/ibex_simple_system.sv` and configuration file `baseline_ibex/ibex_configs.yaml` with the `small` configuration.)

Assistant: High-level response: derive a bottom-up, synth-only filelist for `ibex_simple_system` using the `small` configuration (2-stage pipeline, `RV32MFast`, `RV32BNone`, `RegFileFF`, no ICache, no SecureIbex), and write it as a `.f` file alongside the simple system RTL.

Key configuration parameters for `small` (from `ibex_configs.yaml`):

- `RV32E = 0`
- `RV32M = ibex_pkg::RV32MFast`
- `RV32B = ibex_pkg::RV32BNone`
- `RegFile = ibex_pkg::RegFileFF`

- BranchTargetALU = 0
- WritebackStage = 0
- ICache = 0, ICacheECC = 0, ICacheScramble = 0
- BranchPredictor = 0
- DbgTriggerEn = 0
- SecureIbex = 0
- PMPEnable = 0
- PMPGranularity = 0, PMPNumRegions = 4
- MHPMCounterNum = 0, MHPMCounterWidth = 40

Actions taken (conceptual):

- Examined ibex_simple_system.sv to identify instantiated modules: ibex_top_tracing, local bus, ram_1p, sim_ctrl, timer, and ECC primitives used only when SecureIbex is set.
- Used small config to determine which Ibex features are disabled (e.g. ICache, SecureIbex-only ECC).
- Built a bottom-up module list including only synthesizable modules for this configuration.
- Wrote a synthesis filelist next to the simple system RTL.

Resulting file created:

- baseline_ibex/examples/simple_system/rtl/ibex_simple_system_small.f

Contents of ibex_simple_system_small.f (paths relative to that directory):

```
// Ibex simple system (small configuration) synthesis filelist
// Bottom-up order: leaves first, top-level last.

// Packages & shared primitives
../../rtl/ibex_pkg.sv
../../shared/rtl/prim_assert.sv
../../vendor/lowrisc_ip/ip/prim/rtl/prim_ram_1p_pkg.sv
../../vendor/lowrisc_ip/ip/prim/rtl/prim_ram_1p_adv.sv

// Ibex core leaf blocks
../../rtl/ibex_counters.sv
../../rtl/ibex_alu.sv
../../rtl/ibex_compressed_decoder.sv
../../rtl/ibex_cs_registers.sv
../../rtl/ibex_decoder.sv
../../rtl/ibex_ex_block.sv
```

```
../../rtl/ibex_id_stage.sv
../../rtl/ibex_if_stage.sv
../../rtl/ibex_wb_stage.sv
../../rtl/ibex_load_store_unit.sv
../../rtl/ibex_multdiv_slow.sv
../../rtl/ibex_multdiv_fast.sv
../../rtl/ibex_prefetch_buffer.sv
../../rtl/ibex_fetch_fifo.sv
../../rtl/ibex_pmp.sv
../../rtl/ibex_controller.sv

// Register file (flip-flop based, as per small config RegFileFF)
../../rtl/ibex_register_file_ff.sv

// Tracing core variant used by the simple system
../../rtl/ibex_tracer_pkg.sv
../../rtl/ibex_tracer.sv
../../rtl/ibex_core_tracing.sv
../../rtl/ibex_top_tracing.sv

// Simple system support blocks
./ram_1p.sv
./bus.sv
./sim_ctrl.sv
./timer.sv

// Top-level simple system
./ibex_simple_system.sv
```

This list excludes modules that would not synthesize for the `small` configuration (for example, ICache-related RTL and SecureIbex-only ECC/scrambling blocks).

C.4 Case Study 4: EDA Tool Comparison for an ASIC Design

C.4.1 Exchange 1

User: What ASIC design tools might be useful to me when creating a RISC-V security extension? Can you compare their pros and cons?

Assistant: verbatim Prompt 1: What ASIC design tools might be useful to me when creating a RISC-V security extension? Can you compare their pros and cons?

Response 1: For a RISC-V security extension you'll care about correctness, security properties, and implementability (PPA / physical effects). Different tools hit different parts of that stack, so I grouped them and called out pros/cons plus where they matter.

1. RISC-V-specific design verification

a) ImperasDV (via Synopsys) - A RISC-V processor verification solution with reference models, test suites, coverage, etc. - Useful because it lets you co-simulate RTL

against a golden RISC-V model to catch ISA and microarchitectural bugs early. - Pros: · Ready-made RISC-V reference models and tests · Integrates with major simulators and UVM environments · Used in real RISC-V SoC projects - Cons: · Commercial/expensive · Nontrivial setup

b) Open RISC-V flows: riscv-dv RISC-V Formal - riscv-dv is a UVM instruction generator for random testing. Pros: stress testing, corner cases, privilege transitions. Cons: UVM-heavy, need reference model. - RISC-V Formal checks equivalence between RTL and spec. Pros: open-source, works with Yosys/SymbiYosys. Cons: primary focus on ISA compliance, not custom extensions.

2. Security-focused hardware verification info-flow tools

Cycuity Radix - Hardware security verification platform for information-flow tracking. - Pros: · Security rule libraries · Integrates into simulation/formal flows · Used in aerospace, automotive, data centers - Cons: · Commercial · Requires security-rule learning curve

3. RTL design, simulation debug

Commercial simulators: - Cadence Xcelium, Synopsys VCS, Siemens Questa Pros: full SystemVerilog/UVM support, great performance, strong debug GUIs. Cons: license cost.

Open-source simulators: - Verilator, Icarus, GHDL Pros: free, fast, good for CI. Cons: limited SystemVerilog/UVM support.

4. Formal verification platforms

Cadence JasperGold, Siemens OneSpin 360, Synopsys VC Formal - Pros: · Strong property checking · "Apps" for structural checks, deadlock detection, etc. - Cons: · Requires property-writing expertise · Commercial cost

5. Synthesis, PR, timing power

Synopsys Fusion Compiler / Design Compiler - Industry-leading logic synthesis and physical design. Cadence Genus + Innovus - Widely used, strong physical design integration. Open-source: Yosys + OpenROAD/OpenLane - Pros: fully open. - Cons: PPA not comparable to commercial tools.

6. Power, timing, and side-channel relevance

PrimeTime, Tempus, Voltus, RedHawk-SC - Used for timing signoff, IR drop, power analysis.

7. FPGA / emulation tools

Vivado, Quartus, HAPS, Palladium, Veloce - Useful for prototyping and HW/SW bring-up.

8. Recommended practical stack included: Verilator/Xcelium + riscv-dv, RISC-V Formal, OpenLane or Innovus/Fusion Compiler, FPGA prototyping.

Prompt 2: Could you provide me with some of the more common, industry-level tools?

Response 2: Here is a clean list of common, industry-level EDA tools grouped by stage.

1. RTL Design Simulation: - Cadence Xcelium - Synopsys VCS - Siemens Questa

2. Formal Verification: - Cadence JasperGold - Siemens OneSpin 360 - Synopsys VC Formal

3. Hardware Security Verification: - Cycuity Radix - Synopsys hardware trust tools - Jasper Security Verification apps

4. Synthesis (RTL to gates): - Synopsys Design Compiler - Cadence Genus

5. Place Route: - Cadence Innovus - Synopsys Fusion Compiler

6. Timing, Power, IR drop, SI: - Synopsys PrimeTime - Cadence Tempus - Cadence Voltus - Ansys RedHawk-SC

7. Gate-level simulation: - VCS / Xcelium in GLS modes

8. RISC-V–specific tools: - ImperasDV - Codasip Studio - OpenHW verification environments

9. FPGA/emulation tools: - Synopsys HAPS - Cadence Protium/Palladium - Siemens Veloce

10. Additional infrastructure: - SpyGlass, Verissimo, AscentLint - Git, CI pipelines

11. Minimal "industry core workflow": Design: VCS or Xcelium Verification: UVM, riscv-dv, ImperasDV, JasperGold/OneSpin Security: Radix Implementation: DC, Fusion Compiler/Innovus, PrimeTime Prototyping: HAPS or Palladium

Prompt 3: I'm being provided a license for Cadence tools. Is there any useful information regarding the other options that would still be beneficial to know?

Response 3: Yes — knowing the other ecosystems is valuable.

1. Synopsys VCS vs Cadence Xcelium: - VCS is often fastest for large CPU testbenches. - riscv-dv tends to run fastest under VCS. - Helpful because many CPU/SoC companies standardize on VCS.

2. Fusion Compiler vs Innovus: - Fusion Compiler integrates synthesis + PR and is dominant in high-performance cores. - Innovus has excellent GUI and good integration with Cadence flows. - Knowing both helps when comparing PPA expectations.

3. PrimeTime vs Tempus: - PrimeTime is the signoff timing tool used by most industry teams. - Tempus is very strong and improving quickly. - Understanding PrimeTime concepts is useful even if you can't run it.

4. OneSpin vs JasperGold: - Jasper is excellent and widely used. - OneSpin dominates in safety-critical and ISO26262 markets. - Good to understand their differences conceptually.

5. Cycuity Radix: - Important because Cadence lacks an equivalent native security-verification tool. - Radix performs information-flow and privilege-flow tracking. - Used heavily in secure chip programs.

6. FPGA prototyping: - HAPS (Synopsys) is industry-standard. - Protium (Cadence) integrates well with Palladium. - Knowing HAPS matters for CPU bring-up in industry.

7. Toolchain cultures: - Synopsys: speed, PPA, signoff - Cadence: verification, user-friendly GUIs, academic presence - Siemens: safety, formal rigor, emulation ecosystem

Conclusion: Understanding competing ecosystems is beneficial even when working exclusively in Cadence. It makes you more adaptable, improves design literacy, and maps your work onto real industry workflows.

C.4.2 Exchange 2

User: could you provide me with some of the more common, industry level tools?

Assistant: Absolutely. Here's a clean, industry-level list of the most commonly used ASIC design tools in companies building CPUs, RISC-V cores, and hardware security IP. They are grouped by stage (RTL design, simulation, formal, security verification, synthesis, backend, etc.) with short notes on what each is used for. This is the exact ecosystem you'd see at Cadence, Synopsys, Intel, AMD, NVIDIA, Qualcomm, Apple, SiFive, Northrop, etc.

1. RTL Design and Simulation These are the main tools used to write and debug SystemVerilog RTL. - Cadence Xcelium: widely used, strong debugging (SimVision). -

Synopsys VCS: extremely fast, industry standard for CPU verification. - Siemens Questa: strong assertion support, popular in many flows.

2. Formal Verification Used to mathematically prove properties about pipeline correctness, security logic, and instruction behavior. - Cadence JasperGold: industry gold standard. - Siemens OneSpin 360: strong in safety-critical industries. - Synopsys VC Formal: formal engine integrated into Synopsys ecoystem.

3. Hardware Security Verification Security-specific EDA tools used for secure cores and trust verification. - Cycuity Radix (formerly Tortuga Logic): most widely used tool for information-flow analysis and hardware security verification. - Synopsys Fortify for Hardware: used in some trust flows. - JasperGold Security Apps: formal-based security properties.

4. Synthesis (RTL to Gate-Level Netlist) Converts RTL into timing-accurate implementations. - Synopsys Design Compiler (DC): gold standard for logic synthesis. - Cadence Genus: strong alternative, good Innovus integration. - Siemens Precision: seen in some FPGA/ASIC hybrid flows.

5. Place Route (Physical Design) Tools used to generate real chip layouts. - Cadence Innovus: industry-leading PR with excellent GUI. - Synopsys Fusion Compiler: merges synthesis + PR, strongest PPA. - Siemens Aprisa: newer but growing.

6. Timing, Power, IR Drop, and Signal Integrity Used for signoff-quality timing and power checks. - Synopsys PrimeTime: industry-standard STA. - Cadence Tempus: strong competitor to PrimeTime. - Cadence Voltus: power integrity. - Ansys RedHawk-SC: gold standard for IR drop and thermal analysis.

7. Gate-Level Post-Layout Simulation - Synopsys VCS (GLS mode) - Cadence Xcelium (GLS mode) - Siemens Questa GateSim

8. RISC-V–Specific Verification Tools - ImperasDV: industry-level RISC-V reference model and DV flow. - Codasip Studio: extension design and instruction-set modeling. - OpenHW CV32/CVA6 verification environments (open-source, UVM-based).

9. FPGA Prototyping Emulation Used for pre-silicon testing and firmware bring-up. - Synopsys HAPS: industry-leading FPGA prototype platform. - Cadence Palladium / Protium: high-performance emulation. - Siemens Veloce: competitor to Palladium.

10. Version Control Build Automation (commonly used in ASIC teams) - Git + CI pipelines (GitLab CI, Jenkins) - RTL linting tools (SpyGlass, Verissimo, AscentLint)

Short Industry Rule of Thumb: - VCS is fastest for simulation. - Xcelium is easiest to debug. - JasperGold is top-tier formal, OneSpin rules safety. - Fusion Compiler gives best PPA, Innovus is more user-friendly. - PrimeTime is the timing signoff king. - Radix fills critical security gaps. - HAPS is standard for CPU prototyping.

This list reflects the standard toolchains used throughout the semiconductor industry.

C.4.3 Exchange 3

User: I'm being provided a license for cadence tools, is there any useful information regarding the other options that would be beneficial to know regardless?

Assistant: Yes — even if you are primarily using Cadence tools, there is very valuable knowledge about the other major EDA ecosystems (Synopsys, Siemens, and security-focused tools) that will help you as a hardware designer and especially as someone working on a RISC-V security extension.

1. Synopsys VCS vs. Cadence Xcelium (Simulation) Even if you use Xcelium, most

CPU verification teams also use VCS somewhere. - VCS is often the fastest simulator for large, complex UVM environments. - riscv-dv tends to run extremely fast under VCS. - Many companies maintain VCS-only testbenches.

Knowing how both ecosystems differ helps you transition smoothly between teams.

2. Synopsys Fusion Compiler vs. Cadence Innovus (Physical Design) Fusion Compiler is the industry leader for high-performance PR and integrates synthesis + physical design. - Used heavily by RISC-V startups (SiFive, Ventana). - Very strong PPA (Power, Performance, Area).

Innovus remains preferred for: - Visualization - GUI quality - Academic and mixed flows

Understanding both gives you realistic expectations about physical design differences across companies.

3. PrimeTime vs. Tempus (Static Timing Analysis) PrimeTime is the industry-standard signoff tool. - Even companies using Cadence flows often sign off with PrimeTime. - Many online examples and timing methodologies are written for PrimeTime.

Tempus is excellent and integrates seamlessly with Innovus but knowing PrimeTime concepts makes you industry-literate.

4. OneSpin 360 vs. JasperGold (Formal Verification) - JasperGold is world-class for property checking and security assertions. - OneSpin dominates in safety-critical industries (automotive, aerospace).

Important differences: - OneSpin specializes in "gap detection" and intent validation. - Jasper has stronger app-based workflows (security, X-prop, deadlock).

Understanding both ecosystems helps you speak to verification teams across industries.

5. Hardware Security Tools (Cycuity Radix) Cadence lacks a built-in, industry-standard information-flow analysis tool. Radix is the most widely used hardware security verification platform. - Tracks secret propagation - Flags privilege leaks - Detects unintended information flows

Knowing what Radix does helps you understand the security verification gap in traditional EDA flows.

6. FPGA Prototyping Tools (HAPS vs Protium) - HAPS (Synopsys) is the industry-standard FPGA prototyping platform for CPU design. - Protium is Cadence's FPGA platform and integrates tightly with Palladium emulation.

Understanding HAPS is especially valuable because you will likely encounter it in CPU or SoC bring-up roles.

7. Industry Culture Differences (very important) Synopsys culture: - High-performance design - PPA-driven workflows - PrimeTime signoff

Cadence culture: - Strong verification focus - Better user interfaces - More academic presence

Siemens culture: - Safety-critical flows - Formal verification rigor - Strong emulation (Veloce)

Even if you use Cadence exclusively for your project, understanding these cultures prepares you for real engineering teams.

Conclusion: Understanding competing ecosystems is beneficial even when working entirely within Cadence tools. It expands your design vocabulary, makes you adaptable, and prepares you for professional ASIC workflows where tool diversity is the norm. You become capable of functioning across mixed-tool environments, which is expected in modern

semiconductor design teams.