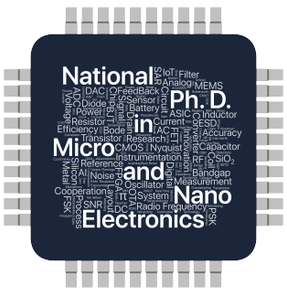

UNIVERSITÀ
DI PAVIA

Design and Implementation of a 28 nm CMOS Bulk RISC-V Single-Core Microprocessor and Peripheral Subsystems

Supervisor: Prof. Marcello De Matteis

PhD Program Coordinator: Prof. Piero Malcovati

Ph.D. Thesis Candidate:
Mirco Malanchini
Sutdent ID 530847

Accademic Year 2024-2025

Contents

List of Figures **iii**

List of Tables **vi**

List of Acronyms **ix**

Abstract **xiii**

1 Introduction **1**

1.1 Aerospace Radiation Effects: Background and Classification . 5

1.2 RISC-V Open-Source Standard 9

1.3 Research Objectives and Thesis Contributions 13

2 Microprocessor Architecture Design and FPGA Prototyping **14**

2.1 High-Level System Macro-Blocks Overview 15

2.2 The *RV32I-Core* Architectural Design 16

2.3 Design of the Hardware *Debug-Module* 23

2.4 Design and Addressing Topology of *Code-Memory* and *Data-Memory* 24

2.5 Architectural Design and Pinout Specification of the *I/O-Module* 26

2.6 FPGA-Based Prototyping and Hardware Verification on Xilinx Spartan-7 28

3 Digital Design Flow and ASIC Implementation **32**

3.1 Standard-Cell Library Characterization and Multi-Corner Multi-Mode (MCMM) Framework 34

3.2 Logic *Synthesis* and Leakage Mitigation Methodologies 37

3.3 Physical Implementation: *Place&Route* Workflow 40

3.4 Post-Layout *Signoff* and *Power-Analyses* Verification 48

3.5 Microprocessor Physical Die Realization and Implementation Parameters 52

4 Verification and Characterization Methodology **55**

4.1 Behavioral Functional-Logic and First-Order Timing Verification 57

4.2 Post-Layout Behavioral Verification, *Signoff* and *Power-Analysis* 59

4.3 Firmware for Instruction-Level Power Characterization 61

5 Experimental Laboratory Setup for Hardware Characterization 69

5.1 Printed Circuit Board Architecture and Design Strategy . . . 72

6 Experimental Results and State-of-the-Art Comparison 75

6.1 *Vector-Less* Statistical Analyses 76

6.2 *Vector-Based* Dynamic Analyses 79

6.3 Experimental Validation and Power Characterization 82

6.4 Simulation Result vs. Post-Silicon Experimental Measurement 84

6.5 Microprocessor Performance Comparison with the State-of-the-Art 87

7 Auxiliary Radiation-Hardened Subsystems and Computational Extensions 90

7.1 Radiation-Hardened-by-Design SRAM Architecture and Verification Methodology. 91

7.2 Compiler-Driven SRAM Integration and Digital Flow Implementation 94

7.3 Floating-Point Unit (FPU) Architecture Integration 98

7.4 Digital Logic Integration and Verification of a 640 MHz Aerospace Phase-Locked Loop (PLL) 101

8 Conclusions and Future Research Lines 107

References 110

A Verilog Source Code of RV32I Microprocessor 119

B Firmware for Microprocessor Characterization 151

List of Figures

1.1 Simulated Single-Event Effects (SEE)-induced substrate current spike characterized by 20 MeV $\cdot$ cm^2/mg Linear Energy Transfer (Linear Energy Transfer (LET)). 7
1.2 Representation of Vulnerability Windows (*Setup* and *Hold* Times) on a Clock Signal. 8

2.1 Top-level block diagram detailing the primary functional macros of the proposed Microprocessor architecture. 15
2.2 Block diagram of the Microprocessor core (*RV32I-Core*), outlining the primary functional units and signal flow. 16
2.3 Official Reduced Instruction Set Computer - Five (RISC-V) RV32I Instruction Set Architecture (ISA) instruction table [1, 2]. 19
2.4 Instruction format topologies and relative decoding field mappings. 20
2.5 Static and dynamic power estimation generated by the Vivado power analysis tool. 31

3.1 Block diagram of the top-down digital flow stage to leakage reduction and radiation hardening. 33
3.2 Pre-synthesis RTL simulation waveforms demonstrating ideal functional timing and zero-delay logic transitions. 39
3.3 Post-synthesis gate-level simulation waveforms illustrating non-ideal cell propagation delays. 40
3.4 Post-*Synthesis* structural netlist showing digital-logic cell clustering. 41
3.5 Core *Floorplan* layout configuration establishing the 0.225 mm square footprint and the dense vertical/horizontal *Power-Grid*. 42
3.6 Standard-cell spatial distribution and legalized placement across core rows prior to *Clock Tree Synthesis*. 44
3.7 Physical layout view of the Microprocessor core after *Clock Tree Synthesis*. 45
3.8 Post-*Routing* physical layout illustrating the fully materialized signal interconnect network and detailed geometric wire paths. 46
3.9 Optimized gate-level routing configuration ready for Graphic Database System II (GDSII) stream-out and *Signoff* verification. 47

3.10 Post-layout timing simulation waveforms exhibiting physical interconnect and cell delay non-idealities. 49
3.11 *Vector-Based* dynamic *IR-Drop* color-coded heatmap showing localized voltage drop distributions across the core *Power-Grid*. 51
3.12 Integrated Microprocessor final layout design (left) and corresponding microphotograph of the physical silicon die (right). . 53

4.1 Block diagram of the hierarchical multi-stage verification framework spanning RTL behavioral simulation, FPGA emulation, and post-layout gate-level ASIC *Signoff*. 57

5.1 Structural block diagram of the post-silicon experimental validation framework, illustrating the connectivity between the *Computer*, Field Programmable Gate Arrays (FPGA) bridge, custom application Printed Circuit Board (PCB), and laboratory instrumentation. 70
5.2 Experimental Measurement and Characterization Setup. 1) *Computer* 2) FPGA (Cmod Spartan A7) 3) Custom PCB 4) DC *Power Supply* 5) *Digital Multimeter* 6) RV32I Microprocessor (DUT - Device Under Test). 71
5.3 PCB Detail of Experimental Measurement and Characterization Setup. 1) RV32I Microprocessor (DUT - Device Under Test) 2) FPGA (Cmod Spartan A7) 3) *Digital Multimeter* Probe 4) RV32I Microprocessor Power Supply 5) PCB auxiliary and Padring Power Supply 6) PLL (Phase-Locked-Loop). 72
5.4 Schematic routing blueprint of the custom PCB, highlighting the bidirectional communication signal paths, level-shifting interfaces, and debug buses interconnecting the FPGA controller and the Device Under Test (DUT). 73

6.1 Post-layout 2D spatial voltage drop map of the Microprocessor, illustrating localized *IR-Drop* profiles under worst-case *Vector-Less* simulation. 78
6.2 Simulated dynamic current consumption profiles across the RV32I ISA, categorized by RISC-V execution tiplogy. 81
6.3 Simulated transient current profile over a 100-clock-cycle window, illustrating continuous clock-cycle power variations ($ADDI$ instruction) and periodic peak modifications induced by the *Loop-Back* branch instruction. 81
6.4 Post-layout dynamic *IR-Drop* spatial distribution map extracted during continuous execution of the $ADDI$ characterization firmware. 82

6.5 Experimental laboratory setup dynamic current consumption profiles across the RV32I ISA, categorized by RISC-V execution tiplogy. 85

7.1 Top-level analog layout view showing the co-integrated standard and Rad-Hard-by-Design (RHBD) Static Random-Access Memory (SRAM) arrays (Left) alongside the central serializer; (Right) Die micrograph highlighting the manufactured silicon macro area. 1) SRAM Rad-Hard, 2) SRAM Digital Serializer and 3) SRAM NO Rad-Hard. 92
7.2 Schematic of the proposed radiation-hardened-by-design SRAM bit-cell featuring an integrated area-neutral decoupling capacitor. 93
7.3 Block scheme of testbench methodology to verify the SRAM. 1) Matlab script 2) Cadence simulation 3) Graphical result analysis. 93
7.4 Graphical User Interface (GUI) of the verification platform displaying automated error mapping and testbench status metrics. 94
7.5 Post-*Place&Route* layout view (Left) and corresponding physical silicon die allocation (Right) of the triple compiled-SRAM subsystem. 1) HVT SRAM, 2) LVT SRAM 3) UHD SRAM. . 97
7.6 Physical transformation of the Floating Point Unit (FPU) layout macro across successive *Place&Route* optimization milestones: (Left) Post-*Place*; (Center) Post-Clock Tree Synthesis (CTS); (Right) Post-*Routing*. 99
7.7 *IR-Drop* voltage degradation map across the consolidated FPU *Power-Grid*. 99
7.8 Finalized GDSII layout macro of the integrated 500 MHz RISC-V FPU subsystem. 101
7.9 Architectural block diagram of the mixed-signal Phase-Locked Loop (PLL) highlighting the structural sub-modules of the synthesized digital logic. 102
7.10 Detailed post-*Synthesis* (Cadence Genus) gate-level schematic of the PLL digital logic. 103
7.11 Post-*Routing* physical layout (left) and corresponding silicon die placement (right) of the 28 nm digital PLL logic, with green boxes highlighting the internal *Probe-PAD* positions for high-frequency testing. 104
7.12 Custom-designed characterization PCB for benchtop laboratory measurement. 105
7.13 Time-domain waveform comparison between the simulated *Feedback Clock* and the empirical benchtop laboratory measurement. 106

List of Tables

2.1 Local register nomenclature defined by Reduced Instruction Set Computer - Five (RISC-V) guidelines. 18
2.2 RV32I instruction set specifications. 22
2.3 Structural field mapping and bit allocation of the debug shift register. 23
2.4 Memory allocation space, register addresses and read/write permissions. 25
2.5 Microprocessor top-level pinout and signal description. 27
2.6 Post-synthesis and post-implementation Look-Up Table (LUT) resource utilization summary. 30

3.1 Standard-cell selection and Multi-Corner Multi-Mode (MCMM) configuration for *Setup* and *Hold* timing analysis. 36
3.2 CAD Ablation Study: Comparison of Unconstrained and Leakage-Mitigated Digital Flows. 38
3.3 Gate Equivalent of Main Components of Microprocessor. . . . 53
3.4 Microprocessor Implementation Parameters. 54

4.1 Assembly testbench code segment for post-layout functional and power-integrity validation. 61
4.2 Firmware characterization loop for the $ADDI$ (*I-Type*) instruction. 63
4.3 Firmware characterization loop for the ADD (*R-Type*) instruction. 64
4.4 Firmware characterization loop for the SW (*S-Type*) instruction. 65
4.5 Firmware characterization loop for the LW (*I-Type*) instruction. 66
4.6 Firmware characterization loop for the BEQ (*B-Type*) instruction. 67
4.7 Firmware characterization loop for the JAL (*J-Type*) instruction. 68

6.1 *Vector-Less* analyses summary parameters and results. 77
6.2 *Vector-Based* analyses summary results. 79
6.3 *Vector-Based* characterization of current consumption for instructions. 80
6.4 Experimental laboratory setup analyses summary results. . . . 83
6.5 Experimental laboratory setup characterization of current consumption for instructions. 84

6.6 Comparison between simulated result and experimentally measurement of the proposed Microprocessor. 85
6.7 Instruction-level dynamic power and energy correlation between simulations results and experimental laboratory setup measurements. 86
6.8 Microprocessor performance comparison with the State-of-the-Art. 87

7.1 Gate Equivalent (Gate Equivalents (GE)) count for each constituent macro-block of the Floating-Point Unit (Floating Point Unit (FPU)). 98
7.2 Architectural parameters and performance of the proposed Floating-Point Unit (FPU). 100
7.3 Architectural parameters and performance of the proposed Phase-Locked Loop (Phase-Locked Loop (PLL)). 104

B.1 Firmware characterization loop for the *ADD* (*R-Type*) instruction. 151
B.2 Firmware characterization loop for the *ADDI* (*I-Type*) instruction. 152
B.3 Firmware characterization loop for the *AND* (*R-Type*) instruction. 153
B.4 Firmware characterization loop for the *ANDI* (*I-Type*) instruction. 154
B.5 Firmware characterization loop for the *AUIPC* (*U-Type*) instruction. 155
B.6 Firmware characterization loop for the *BEQ* (*B-Type*) instruction. 156
B.7 Firmware characterization loop for the *BGE* (*B-Type*) instruction. 157
B.8 Firmware characterization loop for the *BGEU* (*B-Type*) instruction. 158
B.9 Firmware characterization loop for the *BLT* (*B-Type*) instruction. 159
B.10 Firmware characterization loop for the *BLTU* (*B-Type*) instruction. 160
B.11 Firmware characterization loop for the *BNE* (*B-Type*) instruction. 161
B.12 Firmware characterization loop for the *JAL* (*J-Type*) instruction. 162
B.13 Firmware characterization loop for the *Loop-Firmware* instruction. 163
B.14 Firmware characterization loop for the *JALR* (*J-Type*) instruction. 164
B.15 Firmware characterization loop for the *LB* (*I-Type*) instruction. 165
B.16 Firmware characterization loop for the *LBU* (*I-Type*) instruction. 166
B.17 Firmware characterization loop for the *LH* (*I-Type*) instruction. 167
B.18 Firmware characterization loop for the *LHU* (*I-Type*) instruction. 168
B.19 Firmware characterization loop for the *LUI* (*U-Type*) instruction. 169
B.20 Firmware characterization loop for the *LW* (*I-Type*) instruction. 170
B.21 Firmware characterization loop for the *OR* (*R-Type*) instruction. 171
B.22 Firmware characterization loop for the *ORI* (*I-Type*) instruction. 172
B.23 Firmware characterization loop for the *SB* (*S-Type*) instruction. 173
B.24 Firmware characterization loop for the *SH* (*S-Type*) instruction. 174
B.25 Firmware characterization loop for the *SLL* (*R-Type*) instruction. 175

B.26 Firmware characterization loop for the $SLLI$ (*I-Type*) instruction. 176
B.27 Firmware characterization loop for the SLT (*R-Type*) instruction. 177
B.28 Firmware characterization loop for the $SLTI$ (*I-Type*) instruction. 178
B.29 Firmware characterization loop for the $SLTIU$ (*I-Type*) instruction. 179
B.30 Firmware characterization loop for the $SLTU$ (*R-Type*) instruction. 180
B.31 Firmware characterization loop for the SRA (*R-Type*) instruction. 181
B.32 Firmware characterization loop for the $SRAI$ (*I-Type*) instruction. 182
B.33 Firmware characterization loop for the SRL (*R-Type*) instruction. 183
B.34 Firmware characterization loop for the $SRLI$ (*I-Type*) instruction. 184
B.35 Firmware characterization loop for the SUB (*R-Type*) instruction. 185
B.36 Firmware characterization loop for the SW (*S-Type*) instruction. 186
B.37 Firmware characterization loop for the *Blank-Firmware* instruction. 187
B.38 Firmware characterization loop for the XOR (*R-Type*) instruction. 188
B.39 Firmware characterization loop for the $XORI$ (*I-Type*) instruction. 189

List of Acronyms

ABI Application Binary Interface. 18

ADC Analog-to-Digital Converter. 16, 27

AI Artificial Intelligence. xiii, 2, 12

ALU Arithmetic Logic Unit. 17, 79, 80, 83

AMBA Advanced Microcontroller Bus Architecture. 12

ASIC Application-Specific Integrated Circuits. xiii, xiv, 2, 3, 30, 32, 40, 48, 52, 55, 56, 58, 101

AXI4 Advanced eXtensible Interface 4. 12

BOX Buried Oxide. xiii, 2

CAD Computer-Aided Design. 38

CISC Complex Instruction Set Computer. 10

CMOS Complementary Metal-Oxide-Semiconductor. xiii, xiv, 2, 4–6, 13, 32, 34, 37, 44, 52, 54, 75, 77, 79, 82, 83, 85, 92, 101, 104, 107, 108

CPI Clocks Per Instruction. 88

CSR Control and Status Register. 12, 18

CTS Clock Tree Synthesis. v, 4, 44, 45, 91, 98, 99, 103

DAC Digital-to-Analog Converter. 16, 28

DD Displacement Damage. 6

DMA Direct Memory Access. 23

DRC Design Rule Checking. 46–48, 52

DSP Digital Signal Processing. 87

DUT Device Under Test. iv, 58, 70, 73, 74

ECO Engineering Change Orders. 44

EDA Electronic Design Automation. xiv, 3, 13, 35, 38, 48, 50, 51, 56, 57, 59, 60, 62, 69, 70, 75, 78, 84, 86, 95, 96, 98, 103

FD-SOI Fully Depleted Silicon On Insulator. xiii, 2, 5, 6, 107, 108

FoM Figures of Merit. 76, 87

FPGA Field Programmable Gate Arrays. iv, xiii, xiv, 1, 3, 5, 14, 29–31, 56–58, 70–74

FPU Floating Point Unit. v, vii, xv, 4, 91, 98–101, 109

GDSII Graphic Database System II. iii, v, 47, 49, 100, 101

GE Gate Equivalents. vii, xiv, 53, 76, 87, 98, 107

GPIO General-Purpose Input/Output. 70

GPO General-Purpose Output. 16, 27, 28

GUI Graphical User Interface. v, 94

HDL Hardware Description Language. 3, 14, 29, 30, 32, 33, 37, 39, 43, 58, 59, 96, 103

IDE Integrated Development Environment. 29, 57

IoT Internet of Things. 10

IP Intellectual Property. 1, 83

ISA Instruction Set Architecture. iii–v, xiii, 2, 4, 10, 11, 14–16, 19, 20, 52, 55, 56, 58–60, 62, 63, 67, 71, 79, 81, 83, 85–88, 91, 98, 108

JUICE Jupiter ICy moons Explorer. xiii, 2

LDO Onboard Low-Dropout. 73

LET Linear Energy Transfer. iii, 7

LISA Laser Interferometer Space Antenna. xiii, 2

LUT Look-Up Table. vi, 29–31, 58

MCMM Multi-Corner Multi-Mode. vi, 32, 36, 37, 41, 43, 44, 48, 49, 52, 54, 59, 60, 88, 107

ML Machine Learning. xiii, 2, 12

MUSA Multilayered Urban Sustainability Action. xiii, 1, 53

NDA Non-Disclosure Agreement. 44, 95, 96

PC Program Counter. 16, 17, 21

PCB Printed Circuit Board. iv, v, xiv, 70–74, 105, 106

PFD Phase-Frequency Detector. 4, 102, 103

PLL Phase-Locked Loop. ii, v, vii, xv, 3, 4, 74, 91, 101, 102, 104, 109

PNRR National Recovery and Resilience Plan (Piano Nazionale di Ripresa e Resilienza). xiii, 1

PULP Parallel Ultra-Low-Power. 88

PVT Process-Voltage-Temperature. 32, 36

RHBD Rad-Hard-by-Design. v, xv, 4, 9, 91, 92, 109

RISC Reduced Instruction Set Computer. 10

RISC-V Reduced Instruction Set Computer - Five. iii–vi, xiii, xiv, 1–4, 9–20, 23, 26, 52, 55, 58–60, 62, 71, 75–77, 79, 81, 84, 85, 87, 91, 98, 101, 107, 109

RTBD Radiation-Tolerant-By-Design. 26

RTL Register-Transfer Level. xiv, 39, 56–58

SEE Single-Event Effects. iii, 5, 7–9, 32, 37, 40, 43, 72, 90, 108

SEL Single Event Latchup. 8

SET Single Event Transient. 7–9, 78, 90, 102

SEU Single Event Upset. 7–9, 90, 91, 102, 103

SoC System-on-Chip. 109

SRAM Static Random-Access Memory. v, 4, 26, 90–97, 109

STA Static Timing Analysis. 30, 33

STI shallow trench isolation. 6

TASI Thales Alenia Space Italy. xiii, 1

TID Total Ionizing Dose. xiii, 2, 6, 7, 107

TMR Triple Modular Redundancy. 4, 9, 26, 90, 92, 102, 103, 106

TSMC Taiwan Semiconductor Manufacturing Company. 4, 33, 34, 37, 40, 52–54, 91, 94, 95

UART Universal Asynchronous Receiver-Transmitter. 70

VCO Voltage-Controlled Oscillator. 105

Abstract

This thesis presents the comprehensive design procedure, physical implementation, and experimental characterization, via post-layout simulations and laboratory electrical testing, of a 28nm Complementary Metal-Oxide-Semiconductor (CMOS) Bulk Microprocessor based on Reduced Instruction Set Computer - Five (RISC-V) architecture for aerospace applications. This research is carried out within the Multilayered Urban Sustainability Action (MUSA) project, funded by the National Recovery and Resilience Plan (Piano Nazionale di Ripresa e Resilienza) (PNRR), in collaboration with Thales Alenia Space Italy (TASI).

The industrial partner aims to develop Application-Specific Integrated Circuits (ASIC) for different space missions to replace legacy components and reconfigurable logic circuits currently implemented on Field Programmable Gate Arrays (FPGA).

Adopting the 28nm Bulk CMOS process leverages the superior intrinsic radiation resistance regarding Total Ionizing Dose (TID) compared to Fully Depleted Silicon On Insulator (FD-SOI). This choice is critical for medium-to-long-term aerospace missions and large-scale experiments requiring several years (over 10 years) of continuous operation under constant exposure to deep-space radiation. The cumulative TID can reach or exceed 10 MRad for flagship deep-space missions such as Laser Interferometer Space Antenna (LISA), BepiColombo, Jupiter ICy moons Explorer (JUICE), Europa Clipper, and the Parker Solar Probe.

Considering these radiation levels, the FD-SOI technology becomes unsuitable due to its thick Buried Oxide (BOX) layer, which is highly susceptible to significant charge trapping, leading to severe device degradation via back-channel effects. In contrast, the Bulk technology offers a more robust approach for high-dose environments. Although the Bulk process inherently exhibits higher leakage currents compared to FD-SOI, this drawback was effectively mitigated through dedicated design-stage power management and physical optimization.

By leveraging the open-source RISC-V architecture, it is possible to implement a highly customizable hardware platform, thanks to its modular Instruction Set Architecture (ISA) and the ability to integrate custom hardware accelerators for Artificial Intelligence (AI) and Machine Learning (ML).

This programmability allows a single device to replace multiple components, offering a significant advantage given the rigorous aerospace qualification flow (design verification, prototyping, radiation beam testing, and flight qualification) required for aerospace certification.

A specialized digital flow was implemented to enhance radiation hardness and mitigate the static current contribution inherent to Bulk technology.

This work details the Register-Transfer Level (RTL) design and implementation on both FPGA prototypes and ASIC targets. The core is an RV32I (RISC-V Base Integer 32-bit architecture) single-cycle Microprocessor operating at 100 MHz.

Several verification techniques were employed throughout the implementation phases to validate the logical-functional and electrical behavior of the Microprocessor from design to *Signoff*. Given its reprogrammable nature, the verification and characterization of the circuit require a specific standardized testbench and validation strategy. This setup must ensure functional correctness without introducing statistical bias and validate reliability given the prohibitive time and hardware/software resources required for an exhaustive verification of all possible instruction permutations. The validation effort is instead driven towards achieving near-100% functional coverage.

These analyses were conducted through both Electronic Design Automation (EDA) simulations using Cadence tools and experimental measurements on a custom Printed Circuit Board (PCB) in the laboratory. Static (*Vector-Less*) and dynamic (*Vector-Based*) simulations and analyses were performed to characterize the logical-functional and electrical performance.

By leveraging custom-designed testbenches and tailored simulations, the methodology abstracts the hardware verification complexity, encapsulating it into the specific firmware routines used to validate the Microprocessor. This approach enables a direct correlation between simulation results and laboratory data, as the same test routines can be executed in both environments. Furthermore, it facilitates precise electrical characterization, including instruction power consumption, which is fundamental for characterizing complex circuits such as Microprocessors.

The Microprocessor design occupies an area of 0.05 mm^2 (0.225 mm x 0.225 mm), equivalent to 52,682.25 Gate Equivalents (GE) for the integrated circuit (0.010 mm^2, 13,649 GE for the RV32I Core alone), comprising 33,221 digital standard cells and approximately 210,729 CMOS transistors.

The power consumption was evaluated using both *Vector-Less* and *Vector-Based* methodologies. *Vector-Less* analysis verified the circuit behavior under worst-case conditions, assuming a 50 % *Switching-Activity* and a 30 % increase in switching consumption. This resulted in a static current of 0.012 mA (0.44 % static-to-dynamic contribution), a dynamic current of 2.7 mA, and a maximum *IR-Drop* of 39 mV.

The firmware-based dynamic (*Vector-Based*) analyses, conducted at a nominal supply voltage of 0.9 V, showed a simulated static current of 0.011 mA and an average dynamic current per instruction of 1.776 mA. Correlating these results in the laboratory, the experimental measurements demonstrated a de-embedded core static current of 0.01 mA (extracted from a raw measurement of 0.053 mA) and a dynamic current of 1.81 mA, corresponding to a measured energy efficiency of 16.287 pJ/Instruction against the simulated 15.988 pJ/Instruction.

Comparative analysis reveals that the proposed implementation is highly competitive with state-of-the-art solutions, providing a comparable energy-area trade-off and demonstrating the effectiveness of the proposed design methodology for high-reliability applications.

In conclusion, the thesis also presents the design and implementation of peripheral subsystems necessary for the integration of aerospace systems. These include Rad-Hard-by-Design (RHBD) SRAMs, Compiled-SRAMs and Phase-Locked Loop (PLL), as well as a Floating Point Unit (FPU) to enhance computational throughput.

Chapter 1

Introduction

The research work presented in this thesis details the design and physical implementation of a Microprocessor based on the Reduced Instruction Set Computer - Five (RISC-V) RV32I architecture for aerospace applications. Particular emphasis is placed on the integration phase through a digital implementation flow, as well as on the verification and characterization methodologies of the developed Intellectual Property (IP) core via post-layout simulations and laboratory experimental measurements.

This introductory chapter begins by highlighting the context and motivations that drive the presented work. It proceeds by illustrating the primary technological and architectural choices introduced to address the identified challenges. The first section summarizes, at a high level, the topics that will be discussed in the subsequent chapters, outlining the structural guidelines of this thesis. Finally, the chapter concludes with two contextual sections and one section summarizing the overarching objectives of this research work.

The work is framed within the Multilayered Urban Sustainability Action (MUSA) project, funded by the National Recovery and Resilience Plan (Piano Nazionale di Ripresa e Resilienza) (PNRR). MUSA deploys novel paradigms of sustainability and digitalization in response to the challenges arising from the transition of the Milan metropolitan area toward environmental, economic, and social sustainability. Each spoke within the MUSA ecosystem aims to propose and validate technological innovations acting across multiple fronts: economic and financial, environmental (including interurban mobility), and technological.

Within this project, the research was conducted in collaboration with the aerospace company Thales Alenia Space Italy (TASI). The industrial partner aims to develop integrated circuits for diverse aerospace applications to replace legacy components and reconfigurable logic circuits currently implemented on Field Programmable Gate Arrays (FPGA). Beyond meeting specific application use cases, these circuits must exhibit high robustness against radiation-induced damage in the space environment, which inherently degrades and alters the logical-functional and electrical behavior of integrated circuits.

To address these requirements, a Microprocessor-based approach was chosen to provide a highly reprogrammable hardware platform capable of covering multiple application scenarios. The programmability of the Microprocessor allows a single integrated device to replace multiple discrete components, offering a substantial advantage given the rigorous qualification flow (encompassing verification, prototyping, radiation beam testing, and flight qualification) required for aerospace certification.

The adoption of the 28 nm Bulk Complementary Metal-Oxide-Semiconductor (CMOS) process leverages its superior intrinsic radiation tolerance regarding Total Total Ionizing Dose (TID) compared to Fully Depleted Silicon On Insulator (FD-SOI) technology [3–8]. This choice is critical for medium to long term aerospace missions and large-scale experiments requiring several years (exceeding 10 years) of continuous operation under constant exposure to deep-space radiation. The cumulative TID can reach or exceed 10 Mrad [9–11] for flagship deep-space missions such as Laser Interferometer Space Antenna (LISA) [12–15], BepiColombo [16–19], Jupiter ICy moons Explorer (JUICE) [20–24], Europa Clipper [25–30] and the Parker Solar Probe [31–33].

Considering these radiation levels, FD-SOI technology becomes unsuitable due to its thick Buried Oxide (BOX) layer, which is highly susceptible to significant charge trapping, leading to severe device degradation via back-channel effects. In contrast, Bulk technology offers a more robust approach for high-dose environments. Although the Bulk process inherently exhibits higher leakage currents compared to FD-SOI [34–36], this drawback was effectively mitigated through dedicated design-stage power management and physical layout optimization.

The subsequent section will provide the necessary background regarding the effects of aerospace radiation-induced damage (Section 1.1), which is fundamental to understanding the design choices presented throughout this thesis.

By leveraging the open-source RISC-V standard [37–40], it is possible to implement a royalty-free architecture that is highly customizable, thanks to its modular Instruction Set Architecture (ISA) and the flexibility to integrate custom extensions, such as hardware accelerators for Artificial Intelligence (AI) and Machine Learning (ML). These advantages are highly attractive from an industrial and economic standpoint. Furthermore, the standard does not constrain the designer to a specific architectural implementation; this freedom is fundamental for adopting the design and physical implementation techniques required to mitigate radiation effects in aerospace environments. A more detailed overview of the RISC-V standard will be provided in the next section (Section 1.2).

Integrating a Microprocessor into an Application-Specific Integrated Circuits (ASIC) for aerospace applications requires addressing several intermediate

milestones. These steps span from the core design and system integration of the RISC-V architecture to radiation beam testing for hardware hardness validation.

Specifically, the intermediate milestones comprise:

1. Design, implementation, and system integration of a Microprocessor focused on the RISC-V architecture.

2. Design, implementation, and integration of embedded memories and Phase-Locked Loop (PLL), which constitute auxiliary circuits highly sensitive to aerospace radiation effects.

3. Radiation beam testing and verification of the integrated circuits radiation hardness.

4. Co-integration and system-level merger of the tested Microprocessor, memories, and auxiliary circuits.

5. Radiation beam testing and verification of the complete Microprocessor system connected to the memories and auxiliary circuits.

This thesis focuses on the first critical step required for the realization of an aerospace-grade Microprocessor. The final part of the thesis presents the work conducted on the auxiliary circuits (second step) to enhance their radiation tolerance (Chapter 7).

The design of a RISC-V Microprocessor begins with the study of the official standard and its architectural guidelines. An RV32I architecture was designed and integrated, implementing a RISC-V Microprocessor that executes 32-bit integer operations. The core features a single-cycle execution model, operating at a clock frequency of 100 MHz.

The design was developed using a Hardware Description Language (HDL), specifically Verilog, and validated in a behavioral simulation environment. Following the design phase, it was mapped onto an FPGA for logical-functional validation and a preliminary hardware-aware temporal assessment. While accurate timing closure is exclusively achieved through the physical ASIC implementation, FPGA prototyping provides a first-order evaluation of propagation delays and architectural bottlenecks, effectively reducing the abstraction gap inherent to time-agnostic behavioral simulations. During the FPGA implementation stage, official RISC-V tests were executed to verify compliance to the standard.

Once compliance was verified, the Microprocessor was integrated through a digital implementation flow (*Synthesis* and *Place&Route*) using the Cadence Electronic Design Automation (EDA) tool suite, specifically optimized to mitigate static leakage currents and to harden the circuit against radiation-induced damage. At the end of the digital flow, both static (*Vector-Less*) and dynamic (*Vector-Based*) simulations and analyses were performed.

In conjunction with the *Vector-Based* analyses, specific verification techniques were developed to abstract the complexity of the testbench at the firmware level used to stimulate the Microprocessor. Thanks to the integration of a hardware debug module during the design phase and a dedicated laboratory measurement setup, it is possible to execute the exact same testbench in both the simulation environment and on the physical silicon. The implemented verification framework enables a direct correlation and comparison between simulation results and laboratory experimental data. Furthermore, it facilitates precise electrical characterization, with specific granularity on the power consumption per instruction, which is fundamental to properly evaluate a versatile circuit such as a Microprocessor.

The auxiliary circuits consist of a PLL, a Floating Point Unit (FPU) and multiple Static Random-Access Memory (SRAM) blocks, which are fundamental components of a Microprocessor system and are particularly sensitive to aerospace radiation damage. The integrated PLL, implemented in 28 nm CMOS Bulk technology, receives a 40 MHz input reference clock and delivers output clock signals at 40 MHz, 320 MHz, and 640 MHz. Particular detail is given to the work performed on the digital logic specifically the loop counters and the Phase-Frequency Detector (PFD) which was designed using triplication (Triple Modular Redundancy (TMR)) and physical spacing layout techniques to enhance radiation hardness.

Furthermore, a high-frequency FPU operating at a target frequency of 500 MHz has been integrated to natively accelerate single-precision floating-point operations alongside hardware-level multiplication, division, and square root instructions, implementing the standard M and F extensions defined by the RISC-V ISA. Given the aggressive timing constraints and the structural complexity of the mathematical pipeline, special focus is placed on the physical implementation flow. This includes optimization strategies across consecutive back-end milestones namely *Place*, Clock Tree Synthesis (CTS), and *Routing* coupled with the synthesis of a robust, high-density *Power-Grid* layout to strictly contain dynamic voltage degradation (*IR-Drop*).

The described SRAMs were integrated using the 28 nm CMOS Bulk technology node, with specific emphasis placed on the developed verification methodology and their integration with other digital sub-blocks. One SRAM macro was designed using Rad-Hard-by-Design (RHBD) techniques by placing a dedicated capacitance layout structure over the single memory cell. This technique avoids increasing the cell area and does not require cell triplication (thus eliminating the need for voting circuitry) to enhance radiation tolerance. The remaining SRAM blocks were generated using a commercial Taiwan Semiconductor Manufacturing Company (TSMC) SRAM compiler, leveraging the standard cells available in the TSMC technology libraries. These compiler-generated SRAMs are highly optimized in terms of memory density per unit area. The objective

is to evaluate their native radiation tolerance to the degradation effects.

The chapters of this thesis are structured as follows. Chapter 2 describes the architectural design and validation of the Microprocessor in behavioral and FPGA environments. Chapter 3 illustrates the physical implementation and the digital flow along with its specific optimizations. Chapter 4 describes the verification methodologies implemented during the design and integration phases. Chapter 5 details the developed laboratory experimental measurement setup. Chapter 6 reports the results obtained from post-layout simulations and laboratory measurements, benchmarking the performance against the state of the art. Chapter 7 outlines the development of the auxiliary circuits designed concurrently with the Microprocessor to harden the most critical components against radiation. Finally, Chapter 8 draws the conclusions of the research work. The Verilog source codes of the Microprocessor and the verification firmware are provided in Appendix A and B.

1.1 Aerospace Radiation Effects: Background and Classification

This section introduces the primary radiation-induced effects in electronic devices deployed in aerospace environments. The objective is to provide a comprehensive foundation for understanding the degradation mechanisms that can compromise the functional integrity of integrated circuits, with a specific focus on digital subsystems.

In aerospace applications, electronic devices are exposed to diverse radiation sources, including protons, electrons, heavy ions, neutrons, and gamma rays. The interaction of these particles with the constituent materials of an integrated circuit can induce distinct physical phenomena, which are generally classified into two macro-categories: cumulative effects and Single-Event Effects (SEE)s [41–43]. The first effects are linked to the progressive accumulation of damage over time, while the second type is caused by the localized interaction of a single ionizing particle with a sensitive region of the device.

The interaction of ionizing radiation typically generates trapped charges within isolation oxides and defect states at the semiconductor interfaces [44–46]. Consequently, FD-SOI technology exhibits a higher susceptibility to radiation damage due to the intrinsic presence of its characteristic buried oxide layer beneath the channel [3–8]. This vulnerability motivated the selection of a Bulk CMOS process for the physical implementation of the proposed Microprocessor; the absence of a buried oxide layer inherently renders Bulk technology more resilient against these specific radiation-induced degradation mechanisms.

At the microscopic level, these phenomena modify the electrical characteristics of individual transistors, inducing threshold voltage shifts (ΔV_{th}), elevated

subthreshold leakage currents, carrier mobility degradation, degradation of the subthreshold swing, and alterations in the drive currents (I_{on}) [46–48].

In digital circuits, such variations can severely degrade noise margins, increase static power consumption, alter propagation delay times, and, under worst-case conditions, lead to a catastrophic functional failure of the entire integrated circuit.

From a circuit-level reliability perspective, radiation-induced phenomena can be further categorized into *Hard* errors and *Soft* errors. *Hard* errors are associated with permanent physical damage to the device or irreversible degradation of its electrical characteristics, potentially leading to permanent loss of functionality. Conversely, *Soft* errors manifest as transient perturbations of the logic state or the electrical behavior of the circuit, which do not inherently cause permanent physical destruction to the underlying semiconductor structures [49–51].

Hard errors are commonly associated with Displacement Damage (DD), which is primarily driven by non-ionizing energy loss from particles such as neutrons or high-energy protons interacting with the semiconductor crystal lattice. These interactions displace atoms from their nominal lattice sites, creating interstitial vacancy pairs and stable crystalline defects that alter the material properties. This type of damage reduces carrier mobility, shortens minority carrier lifetimes, and severely degrades the performance of sensitive devices like active pixel sensors, photodiodes, and bipolar junction transistors [52].

On the other hand, *Soft* errors are non-destructive transient faults triggered by single ionizing particles that generate localized current pulses, temporarily perturbing the logic state without inflicting physical damage.

The cumulative parametric degradation of the device is governed by the Total Ionizing Dose (TID), which quantifies the total ionizing energy absorbed by the device per unit mass over its operational lifespan. Unlike single-event phenomena, TID induced degradation does not stem from an instantaneous interaction but from the progressive accumulation of trapped holes in the insulating layers of the transistor and the generation of interface states. Critical regions susceptible to TID include the thin gate oxide, shallow trench isolation (STI) oxides, spacer oxides, and the silicon-silicon dioxide (Si/SiO_2) interfaces [44–47].

To enhance circuit-level resistance to TID, the technology selection is foundational, such as opting for Bulk CMOS rather than FD-SOI. Furthermore, precisely because TID accumulation induces severe parametric shifts akin to extreme process variations, it is mandatory during the physical implementation phase to validate circuit robustness across worst-case boundary conditions via comprehensive corner and Monte Carlo statistical analyses. This is crucial because TID effects progressively shift the circuit's electrical operating point to-

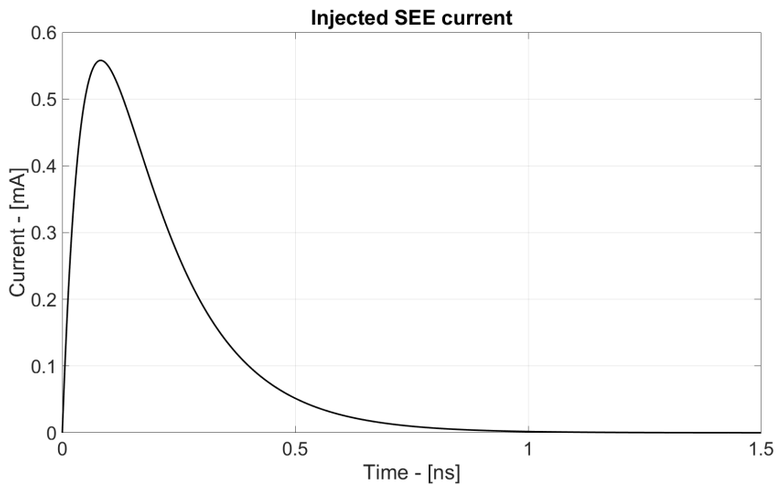

Figure 1.1: Simulated SEE-induced substrate current spike characterized by 20 MeV $\cdot$ cm^2/mg Linear Energy Transfer (LET).

ward these worst-case operational boundaries over time. Furthermore, because charged particles deposit charge into the shared substrate, the introduction of deep N-Well isolation pockets around the transistors intrinsically enhances the circuit's radiation hardness [53–55].

In addition to TID, SEEs represent another critical class of radiation-induced phenomena. SEEs are triggered by the passage of a single ionizing particle through a sensitive node of the device. The energy deposited along the particle's track generates electron-hole pairs that are subsequently collected by the electric fields of circuit nodes, producing transient current pulses or unintended logic state transitions [56–58].

The incident particles deposit their charge within the circuit substrate, generating transient current spikes (Figure 1.1). The magnitude of this current pulse depends heavily on the specific incident particle and is directly proportional to its LET. In space environments, these current spikes can reach peak amplitudes of 0.55 mA with transient durations spanning a few hundred picoseconds, assuming a nominal LET of 20 MeV$\cdot$cm^2/mg [56,59,60]. Depending on the specific circuit node struck by the ionizing radiation, this injected charge can provoke a small-signal voltage oscillation whose amplitude and settling time are governed by the node's equivalent impedance and resistivity.

In digital circuits, SEEs are particularly critical because they can directly corrupt the data being processed or stored.

These temporary current spikes manifest as transient events. In digital domains, their impact varies depending on whether they strike combinational logic or sequential elements. When a SEE disrupts a sequential component, it is classified as a Single Event Upset (SEU). An SEU can induce an unwanted change of state within the sequential element, causing a bit-flip ($0 \rightarrow 1$ or $1 \rightarrow 0$). On the other hand, a Single Event Transient (SET) occurs when a SEE strikes a combinational logic block, generating a transient voltage glitch.

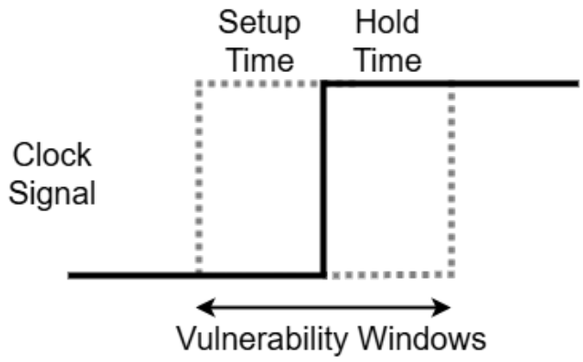

Figure 1.2: Representation of Vulnerability Windows (*Setup* and *Hold* Times) on a Clock Signal.

SETs only temporarily perturb the circuit's behavior, as the combinational logic will naturally propagate the glitch and re-establish the correct static logic state. However, if a SET arrives at a sequential element exactly during its synchronous *Setup* and *Hold* vulnerability window (Figure 1.2), the corrupted transient value will be latched, thereby propagating and transforming into an SEU [57, 58].

Equally critical is the scenario where a SET strikes the global clock or reset networks, as a transient glitch on these lines can trigger false clock edges or unintended system initializations. To mitigate this specific vulnerability, the adoption of synchronous resets is generally preferred over asynchronous implementations, since synchronous inputs are evaluated strictly at the clock edge, thereby intrinsically masking transient glitches.

Among the most severe SEEs are Single Event Latchup (SEL)s. An SEL represents a more critical scenario, as it can activate parasitic structures internal to the CMOS circuit, generating high currents that, if not interrupted, can lead to permanent damage to the device [55, 61].

Within digital architectures, memory elements (such as latches and D-flip-flops) constitute the most vital sequential components and represent the most radiation-sensitive regions in aerospace environments. If an SEU induces a bit-flip within a memory cell, it corrupts the stored data. During execution, this corrupted state propagates through the system, resulting in inconsistent computations or system crashes [62–64].

To prevent such error propagation, architectural mitigations leveraging temporal and spatial redundancy can be deployed. Redundancy techniques typically involve triplicating the critical circuit and introducing voting circuits (majority voters) to determine the correct logical output. Temporal redundancy triplicates the execution over time, requiring the same computation to be performed three times at distinct temporal intervals. This ensures that a transient current spike can disrupt at most one of the three computational cycles. The required temporal spacing between execution intervals depends on the maximum expected duration of the radiation-induced current pulse

within the specific mission profile. While temporal redundancy enhances radiation hardness, it severely penalizes performance, restricting the maximum operating frequency to approximately one-third (1/3) of the non-triplicated baseline [65–67].

In contrast, spatial redundancy multiplies the physical hardware components, allowing the triplicated computation to be executed concurrently in parallel. During the physical *Place&Route* phase, these triplicated elements must be placed with a minimum spatial separation distance to prevent a single particle track from simultaneously upsetting multiple redundant nodes. Spatial redundancy preserves the maximum clock frequency of the circuit but incurs severe penalties in terms of silicon area and power consumption [65–67].

When an SEU originates from an SET captured during its vulnerability window, it becomes highly challenging to detect and manage because the corrupted data is directly written into the element as a valid latching event rather than manifesting as a spontaneous post-latching bit-flip. Standard redundancy techniques manage errors within already stored data but do not offer complete protection during the exact sampling phase of a new data input. This sampling occurs at the output of the combinational path, right where voting circuits cannot be easily interjected without adding further vulnerability (typically at the output of the voting circuits) [65–70].

Consequently, while Triple Modular Redundancy (TMR) is a powerful tool to increase radiation tolerance, it must be complemented by RHBD layout techniques during the physical design phase to provide intrinsic hardware-level immunity. These physical implementation techniques, which are detailed in subsequent chapters (Chapter 3), must explicitly account for radiation-induced current spikes that cause voltage fluctuations on the logic nodes [71–73].

An excessive voltage fluctuation exceeding 25% of the nominal supply voltage (V_{DD}) can easily induce an unintended bit-flip. To mitigate this effect, it is mandatory to implement a highly dense Power Grid architecture capable of sinking the current transients without local rail collapse. At the *Signoff* phase of the digital flow, static and dynamic IR-drop analyses are conducted to verify the maximum voltage drop across the power network during operation. If the native IR-drop is high, its superposition with a SEE induced current spike can easily cause functional bit-flips or computational logic errors [74–76].

1.2 RISC-V Open-Source Standard

This section introduces the open-source RISC-V architectural standard, delineating its historical evolution, modular framework, and structural advantages over proprietary alternatives.

The RISC-V (Reduced Instruction Set Computer, Fifth Generation) project originated in 2010 at the University of California, Berkeley. Initially conceived

by a research team led by Krste Asanović, Andrew Waterman, and Yunsup Lee, alongside David Patterson (one of the foundational pioneers of the RISC paradigm). The architecture was developed to support academic research and development in digital systems without the constraints of proprietary constraints. Recognizing its immense industrial potential, the creators established the RISC-V International foundation (formerly the RISC-V Foundation) to govern, standardize, and promote the ISA. Over the past decade, interest in the RISC-V model has expanded exponentially, transitioning from an academic tool into a dominant commercial and scientific standard widely adopted in industrial automation, Internet of Things (IoT), and high-reliability aerospace applications [77–79].

The fundamental characteristic that distinguishes Microprocessors is their ISA, which defines the boundary between hardware and software by specifying the supported computational operations, register structures, and instruction encoding formats. Historically, ISAs have been categorized into two main philosophical design paradigms: Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC) [78, 79].

- **CISC Architecture (e.g., x86, AMD64):** Predominant in traditional desktop and server computing, CISC was historically driven by the limitations of early memory density. It is characterized by variable-length, highly complex instructions. A single CISC instruction can encapsulate multiple lower-level operations abstracted via hardware-level microcode such as simultaneous memory accesses and arithmetic-logic computations. While this approach minimizes the overall code footprint of executable binaries, it demands a massive, highly complex control unit and execution hardware. This structural complexity compromises power efficiency and clock frequency potential, rendering CISC architectures less suitable for strict power-constrained environments [78, 79].

- **RISC Architecture (e.g., ARM, RISC-V):** RISC philosophy prioritizes hardware simplicity and architectural efficiency. It features a streamlined set of simple, fixed-length instructions (typically 32 bits), each executing a singular, well-defined task within a predictable number of clock cycles. Although a RISC-based software program requires a higher total instruction count and a larger memory footprint to accomplish the same computational task as a CISC equivalent, the underlying physical hardware is significantly smaller, faster, and more energy-efficient [78, 79].

While proprietary RISC architectures like ARM have long dominated the mobile and embedded markets where power efficiency is paramount, the market has recently witnessed a broader migration toward RISC processors in high-performance laptop and server domains, driven by their superior performance-per-watt metrics.

In the contemporary semiconductor market, the most commercially successful ISAs namely x86/AMD64 and ARM are proprietary. Utilizing these commercial architectures requires semiconductor design teams to acquire prohibitively expensive licenses, enter complex non-disclosure agreements, and pay continuous royalties per device shipped. Furthermore, these proprietary standards tightly restrict or completely forbid any custom modifications to the underlying hardware architecture.

In contrast, RISC-V is a completely open-source, royalty-free standard available globally without licensing fees. From an industrial and economic perspective, this model drastically reduces development costs and democratizes application-specific silicon design. Furthermore, the open-source nature fosters an expansive global ecosystem of comprehensive documentation, verified open-source hardware blocks, and an abundance of peer-reviewed scientific literature covering analog, digital, and mixed-signal designs, proving the vibrant activity of the scientific community in this domain [1, 2, 77–79].

Unlike proprietary ISAs, RISC-V fundamentally separates the ISA specification from its architectural implementation. The standard acts strictly as a functional contract: it defines the logical-mathematical behavior expected for each instruction, leaving the physical design choices (e.g., pipeline depth, execution units, power mitigation) entirely to the discretion of the digital engineer. This structural freedom is crucial for implementing custom hardware-level mitigations against radiation-induced damage without violating standard toolchain compliance [1, 2, 77–79].

The defining structural feature of the RISC-V standard is its strict modularity. Rather than enforcing a monolithic, over-engineered instruction set where many specialized operations remain unused by generic software, RISC-V enforces a minimal, frozen base integer instruction set. This base integer core is mathematically Turing-Complete, satisfying the Jacopini-Böhm theorem by natively supporting the foundational control flow mechanisms sequence, selection (conditional branching), and iteration (loops) required to execute any computable algorithm [78–80].

The standard defines three primary base architectures categorized by the native bit-width of the internal general-purpose registers, the data word, and the memory address space [1, 2, 78, 79]:

- **RV32:** A 32-bit architecture utilizing thirty-two 32-bit registers, highly optimized for embedded systems, microcontrollers, and resource constrained accelerators.
- **RV64:** A 64-bit architecture tailored for high-performance application processors and desktop-class operating systems.
- **RV128:** A 128-bit architecture designed for massive warehouse-scale computing and high-capacity memory mapping.

- **RV32E:** The standard also specifies a reduced variant, which includes only 16 general-purpose registers, targeting ultra-low-power, deeply embedded microcontrollers.

To expand the capability of the base integer core, designers can modularly layer standard, well-defined instruction extensions. The standard software compiler toolchain guarantees out-of-the-box compliance for these extensions [1, 2, 78, 79]:

- **I (Integer):** The mandatory base integer instruction set (e.g., RV32I). Implements standard integer arithmetic, logic, and control flow.
- **M (Multiply):** Appends hardware-level integer multiplication and division operators.
- **A (Atomic):** Introduces atomic read-modify-write memory operations. Essential for multi-core synchronization, preventing ABA synchronization errors, and supporting advanced system bus protocols such as Advanced Microcontroller Bus Architecture (AMBA) and Advanced eXtensible Interface 4 (AXI4).
- **F (Float):** Integrates a dedicated single-precision floating-point register file and corresponding computational, load, and store instructions.
- **D (Double):** Extends the floating-point subsystem to support double-precision floating-point data and operations.
- **Zicsr:** Implements the Control and Status Register (CSR), vital for handling hardware interrupts, exceptions, performance monitoring, and fault management.

Beyond these standard modules, RISC-V allocates a dedicated opcode space for custom extensions. This allows the seamless integration of proprietary hardware accelerators such as specific AI and ML vector engines provided that the designer manages the preliminary integration steps to avoid hardware conflicts with the existing standard extensions [1, 2, 78, 79].

All base RISC-V instructions occupy a fixed length of 32 bits and must be aligned to 32-bit memory boundaries. To maximize hardware efficiency and simplify decoding logic, the standard maintains a highly symmetric instruction layout [1, 2, 78, 79].

Additionally, the RISC-V assembly language defines pseudoinstructions mnemonic shorthand solutions that simplify software development. *Pseudoinstructions* do not expand the hardware instruction set; instead, the compiler toolchain automatically translates them into their standard hardware equivalents during compilation [1, 2, 78, 79].

1.3 Research Objectives and Thesis Contributions

The research work described in this thesis aims to design and physically integrate a RISC-V Microprocessor utilizing a 28 nm CMOS Bulk technology node. Particular emphasis is placed on the digital implementation flow and the complex verification and characterization frameworks mandatory for high-reliability, general-purpose integrated circuits.

Achieving this milestone represents the foundational first step toward developing a Microprocessor core with robust radiation hardness, thereby enabling its deployment in harsh aerospace applications.

The experimental results obtained from both post-layout EDA simulations and laboratory electrical measurements validate the successful achievement of these research objectives. Furthermore, benchmarking against the state of the art demonstrates that the proposed Microprocessor implementation is highly competitive, yielding superior performance and energy-area trade-offs compared to the average of comparable baseline architectures documented in recent literature.

Chapter 2

Microprocessor Architecture Design and FPGA Prototyping

This chapter describes the architectural design of the proposed Reduced Instruction Set Computer - Five (RISC-V) Microprocessor. The processor implements the RV32I Instruction Set Architecture (ISA), utilizing a single-cycle execution model operating at a clock frequency of 100 MHz [81].

Adopting a top-down design methodology, the complete architecture is first examined at the system level, illustrating the interconnected macro-blocks: the *Debug-Module*, the *RV32I-Core*, the *Code-Memory*, the *Data-Memory*, and the *I/O-Module*. Subsequent sections detail the architecture of each individual subsystem, concluding with the hardware implementation and mapping of the Microprocessor onto a Field Programmable Gate Arrays (FPGA) platform.

Each macro-block is characterized by its logical behavior and its specific functionality within the Microprocessor, detailing its primary constituent components. For comprehensive design verification, the complete Verilog source code developed to describe the Microprocessor is provided in Appendix A.

The Microprocessor architecture was modeled using Verilog, a standardized Hardware Description Language (HDL). the HDLs language implement circuit design through the formal description of the hardware's logical-functional behavior. This description allows the system to be defined down to the level of a single primitive logic gate. To enhance expressiveness, these languages leverage the primitives and constructs typical of imperative-procedural programming languages.

However, HDL fundamentally differ from standard software languages because they are intrinsically concurrent, mirroring the physical parallelism of hardware structures. Unlike sequential software execution paths, digital circuits operate in parallel and follow an event-driven mechanism, typically synchronized by the rising-edge of a clock signal that governs the entire system. Consequently, HDL implement an imperative-behavioral paradigm.

To integrate a digital circuit onto silicon, the code must describe a parallel

(concurrent) behavior, defining its response (output signals) to the events (input signals) that trigger it (rising-edge of the clock signal).

2.1 High-Level System Macro-Blocks Overview

This subsection introduces the macro-blocks that constitute the Microprocessor architecture, presented through a top-level block diagram.

The implemented Microprocessor based on the RISC-V architecture features an RV32I single-cycle ISA. Specifically, it executes integer operations utilizing a 32-bit data representation at an operational clock frequency of 100 MHz.

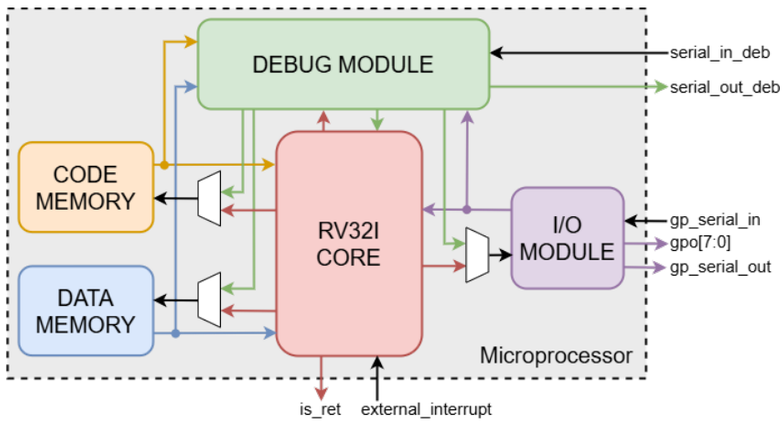

Figure 2.1: Top-level block diagram detailing the primary functional macros of the proposed Microprocessor architecture.

Figure 2.1 illustrates the macro-block schematic of the designed system. The overall architecture integrates a dedicated RISC-V core (*RV32I-Core*), a hardware debug unit (*Debug-Module*), two synchronous memories (*Code-Memory* and *Data-Memory*), and a peripheral interface block designed to interface the Microprocessor with external components (*I/O-Module*).

- ***RV32I-Core*:** The core execution engine of the Microprocessor, implementing a single-cycle hardware design compliant with the standard RISC-V RV32I ISA specification.
- ***Debug-Module*:** A fundamental hardware component designed to monitor and control the Microprocessor architecture during execution, enabling full runtime verification of its logical and functional behavior.
- ***Code-Memory*:** A synchronous memory block implemented via D-flip-flops, structured as a matrix of 32 rows by 32 columns. This configuration provides a total capacity of 128 bytes dedicated to storing the instructions of the executable firmware binary.
- ***Data-Memory*:** Similarly implemented as a synchronous memory block but organized as a matrix of 128 rows by 8 columns. It provides a total

allocation space of 128 bytes used for storing variables, datasets, and information processed during firmware execution.

- ***I/O-Module*:** The primary peripheral subsystem enabling external hardware communication. It integrates two synchronous serial interfaces, suitable for connecting to external (for example, Digital-to-Analog Converter (DAC) and Analog-to-Digital Converter (ADC)), alongside 8 dedicated General-Purpose Output (GPO) ports.

2.2 The *RV32I-Core* Architectural Design

This subsection details the *RV32I-Core* macro-block, which represents the processing core of the designed Microprocessor.

The RISC-V core implements the RV32I ISA. The ISA serves as the fundamental abstract interface between the hardware and the firmware, decoupling the physical hardware design from a specific software implementation. It defines the mandatory instruction set that must be implemented within the Microprocessor to ensure full compliance with the standard, mathematically characterizing the logical-arithmetic function of each operation.

The RV32I ISA comprises integer operations (the standard *I* extension) featuring a 32-bit binary data representation. The implemented Microprocessor employs a single-cycle execution model, meaning it executes exactly one instruction per clock cycle, targeting an operational clock frequency of 100 MHz.

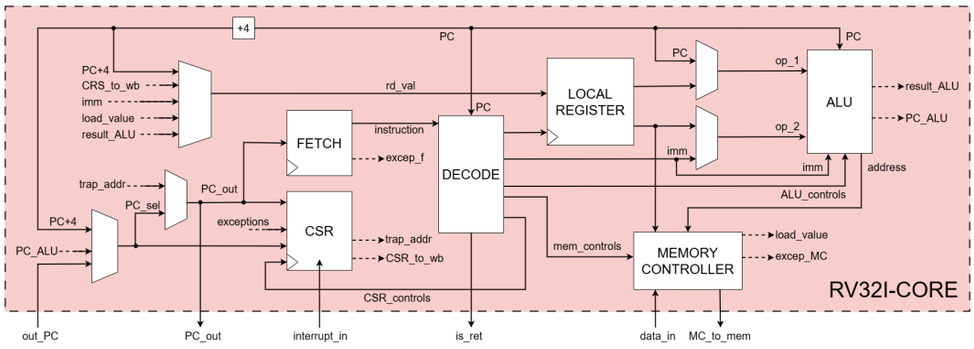

Figure 2.2: Block diagram of the Microprocessor core (*RV32I-Core*), outlining the primary functional units and signal flow.

Figure 2.2 show the qualitative top-level block diagram of the *RV32I-Core* macro-block. The RISC-V standard does not impose rigid structural constraints on the physical implementation; instead, it provides open guidelines and design best practices to be followed during hardware engineering. To process each instruction, the standard establishes three primary logical phases:

- **Fetch:** Based on the memory pointer stored in the Program Counter (PC), the system reads the content of the memory location within the

Code-Memory containing the firmware binary. The targeted memory register outputs the 32-bit instruction word, routing it directly to the *Decode* stage.

- **Decode:** Given the retrieved 32-bit instruction word, this phase decodes the specific bit-fields according to the standard, subsequently activating the corresponding control and data-path signals within the Microprocessor to execute the identified instruction.

- **Execute:** Once the instruction is decoded and the respective core datapath segments are asserted, this phase performs the actual computation or control operation.

Because this is a single-cycle Microprocessor, all three logical phases are executed within a single clock cycle to process the active instruction fetched from the *Code-Memory*.

The block diagram in Figure 2.2 clearly distinguishes the dedicated hardware components associated with the three execution phases described above. The Fetch and Decode stages are managed by the homologous *FETCH* and *DECODE* architectural blocks, whereas the Execute phase is primarily executed by the Arithmetic Logic Unit (ALU).

The block diagram (Figure 2.2) also illustrates the other primary components mandated by the RISC-V standard:

- **Program Counter (PC):** The memory pointer targeting the *Code-Memory* that defines the address of the current instruction scheduled for execution.

- **ALU:** The core computational engine that executes all logical and mathematical operations. It calculates the effective memory addresses for data storage (*S-type* Store instructions) and retrieval (*I-type* Load instructions). Furthermore, it evaluates the target address for instructions that directly modify the program flow (*J-type* jumps, *B-type* conditional branches, and *U-type* instructions).

- **Local Register:** A synchronous memory structure composed of 32 words (matrix of 32 rows by 32 bits). Representing the fastest storage array closest to the ALU, it holds the immediate operands required for computational processing. The instructions access this register file via dedicated source specifiers to fetch the operands needed for execution.

- **Memory Controller:** A dedicated control block that manages the handshaking and data transfer with the external *Data-Memory* during load and store operations.

- **Control and Status Register (CSR):** This component provides the necessary hardware hooks for interrupt management and exception handling. Interfacing with the *Debug-Module*, the CSR block can halt the execution of the Microprocessor, transferring control to the external debugger to enable comprehensive verification and state control of the core.

Table 2.1: Local register nomenclature defined by RISC-V guidelines.

Name	**Alias**	**Role**	**Save**
X_0	zero	Hardwired 0	No
X_1	ra	Return Adress	Yes
X_2	sp	Stack Pointer	No
X_3	gp	Global Pointer	No
X_4	tp	Thread Pointer	No
X_5 - X_7	t0 - t2	Temporary	No
X_8 - X_9	s0 - s1	Saved	Yes
X_{10} - X_{15}	a0 - a5	Arguments	No
X_{16} - X_{17}	a6 - a7	Arguments	No
X_{18} - X_{27}	s2 - s11	Saved	Yes
X_{28} - X_{31}	t3 - t6	Temporary	No

The RISC-V standard defines the *Local Register* as a synchronous memory bank containing 32 registers (32-bit words for the RV32I architecture). It enforces a strict architectural constraint on the *x0* register, which must maintain a constant logical value of zero. Consequently, *x0* acts as a hardwired zero-sink (or null register), providing a stable reference value and facilitating the initialization or clearing of other general-purpose registers.

Table 2.1 specifies the exact nomenclature of the register contained in the *local register*. While the standard prescribes specific software conventions and best practices for register utilization to maximize software toolchain and compiler compatibility, the hardware allows arbitrary use of registers *x1* through *x31*. The only immutable structural constraint remains enforced on *x0*.

The *Alias* column in table 2.1 indicates the name of the registers (Application Binary Interface (ABI)), the *Role* column defines their standard usage, and the *Save* column specifies whether the register contents must be preserved into the *Data-Memory* before being overwritten during a function call.

The register allocation best practices defined by the RISC-V standard are structured as follows:

- ***x1* or *ra*:** Stores the return address, serving as the target destination for unconditional link instructions (*JAL* and *JALR*).

- ***x2* or *sp*:** Functions as the stack pointer, holding the active address

boundary of the stack within the *Data-Memory*, dynamically incrementing or decrementing based on memory allocation and deallocation routines.

- ***x3* or *gp*:** Utilized as the global pointer to access static variables.
- ***x4* or *tp*:** Functions as the thread pointer in multi-threaded software environments.
- ***x8* or *fp*:** Functions as the frame pointer, defining a specific segment of the *Data-Memory* allocated to preserve function arguments and local variables.
- ***a0*–*a7*:** Designated for passing function arguments; specifically, *a0* and *a1* are also utilized to return function results to the calling routine.
- ***t0*–*t6*:** Designated to hold temporary, volatile variables that do not need to be preserved across function calls.
- ***s0*–*s11*:** Designated as saved registers that must be preserved across function boundaries.

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Figure 2.3: Official RISC-V RV32I ISA instruction table [1, 2].

To formally define a compliant RISC-V Microprocessor architecture, the specific underlying ISA implementation must be detailed. Figure 2.3 summarizes all 40 instructions comprising the baseline RV32I ISA. Among these, 37 instructions execute logical-mathematical computations and control-flow modifications, whereas the remaining 3 instructions (*FENCE*, *ECALL*, and *EBREAK*) serve as execution control primitives. Specifically, these system instructions halt or fence standard firmware execution to yield control back to the hardware debugger or operating system environment.

As illustrated in Figure 2.3, each individual instruction is mapped to a specific subset of the available 32 bits dictated by the standard encoding. These patterns define sub-groups associated with the logical-mathematical functions of the instructions, which are directly mapped into the 6 primary format types defined by the RISC-V standard: *R-type*, *I-type*, *S-type*, *U-type*, *B-type* and *J-type*.

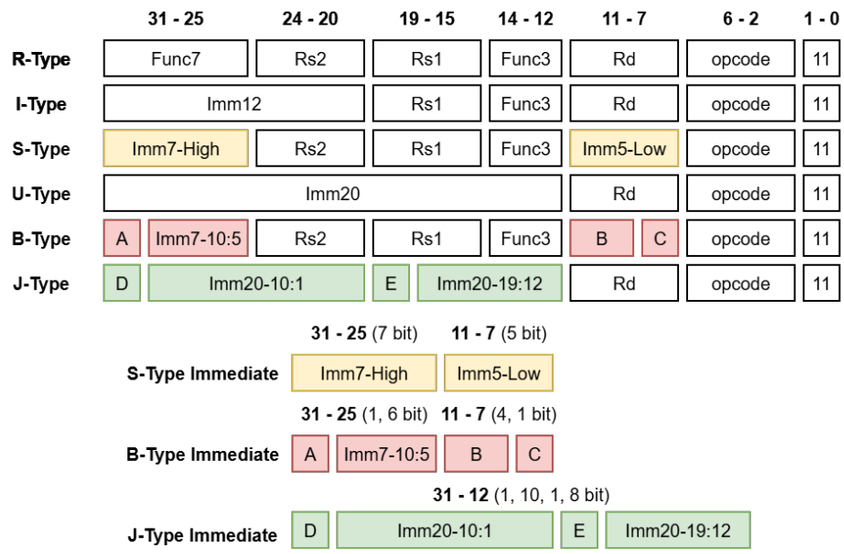

Figure 2.4: Instruction format topologies and relative decoding field mappings.

Figure 2.4 summarizes these instruction formats and their respective bit-field topologies. Every instruction is uniquely characterized by an *opcode* field, which specifies the structural category and primary operation type to which it belongs.

The 6 mathematical and logical instruction formats are defined as follows:

- ***R-Type* (Register):** Implements register-to-register operations. These instructions execute logical-mathematical operations between two source registers located within the local register file (source registers *Rs1* and `Rs2`) and store the computed result into a target register (destination register *Rd*). The source and destination registers can either be distinct or point to the same physical register location.

- ***I-Type* (Immediate):** Implements register-to-immediate operations. These instructions execute logical-mathematical operations between a

single source register (*Rs1*) and a sign-extended 12-bit immediate constant (*Imm12*), writing the output into a destination register (*Rd*). The immediate constant embeds the literal numerical value directly within the instruction word, bypassing the need to fetch it from a memory address pointer. This format also encompasses specialized memory Load operations and the unconditional indirect jump instruction (*JALR*). Load instructions read data from a specific memory location within the *Data-Memory* and write it into the register file, while *JALR* alters the program flow by directly overwriting the Program Counter.

- ***S-Type* (Store):** Implements memory write operations to export data to the system. The source registers (*Rs1* and *Rs2*) act as data and address pointers: *Rs1* identifies the register holding the data to be written, whereas the target address within the *Data-Memory* is computed by adding the base address in *Rs2* to a split 12-bit immediate offset (composed of *Imm7-high* and *Imm5-low*).

- ***U-Type* (Upper Immediate):** Implements long-immediate loading to construct 32-bit constants. These instructions load a 20-bit immediate value into the upper bits of the destination register *Rd*, which is fundamental for enabling long-range unconditional jumps to distant memory regions relative to the current Program Counter.

- ***B-Type* (Branch):** Implements conditional branching operations that redirect the Program Counter if a specific logical condition is satisfied. The hardware compares the values stored in source registers *Rs1* and *Rs2*; if the conditional test evaluates to true, the Program Counter is updated by a signed 12-bit offset computed from a scrambled immediate field (composed of *A*, *Imm7*, *B* and *C* to optimize hardware routing).

- ***J-Type* (Jump):** Implements unconditional long-range jumps. These instructions save the sequential return address ($PC + 4$) into the destination register *Rd* to facilitate subroutine returns, while simultaneously updating the Program Counter with a signed 20-bit offset calculated from the immediate fields (composed of *D*, *Imm20-10:1*, *E* and *Imm20-19:12*).

Depending on the format, the 32-bit instruction word decodes into distinct functional fields (Figure 2.4):

- ***Rd*:** Destination register specifier, encoded as a 5-bit field.

- ***Rs1*:** First source register specifier, encoded as a 5-bit field.

- ***Rs2*:** Second source register or target data register (for *S-Type* operations), encoded as a 5-bit field.

- ***Func3* and *Func7*:** Additional operation specifiers nested within an *opcode* category, encoded as 3-bit and 7-bit fields, respectively.
- ***Imm12*:** A 12-bit literal immediate value utilized in *I-Type* operations.
- ***Imm7-High* and *Imm5-Low*:** Split immediate fields of 7 bits (most significant) and 5 bits (least significant), which are concatenated by the hardware to form the 12-bit immediate value for *S-Type* operations.
- ***Imm20*:** A 20-bit literal upper immediate value utilized in *U-Type* operations.
- ***A, Imm7-10:5, B, C*:** Discontinuous bit fields (1 bit, 6 bits, 4 bits, and 1 bit, respectively) concatenated by the decode logic to reconstruct the 12-bit signed offset for *B-Type* conditional branches.
- ***D, Imm20-10:1, E, Imm20-19:12*:** Discontinuous bit fields (1 bit, 10 bits, 1 bit, and 8 bits, respectively) concatenated by the decode logic to reconstruct the 20-bit signed offset for *J-Type* unconditional jumps.

Table 2.2: RV32I instruction set specifications.

Name	Extended Name	Instruction Type	Logical-Mathematical Functions	Example	Operands
LUI	Load Upper Immediate	U	R[rd] = sext(imm[31:12] <<12)	lui x5, imm	rd (5 bit); imm (20 bit)
AUIPC	Add Upper Immediate to PC	U	R[rd] = PC + sext(imm[31:12] <<12)	auipc x5, imm	rd (5 bit); imm (20 bit)
JAL	Jump and Link	J	R[rd] = PC + 4; PC = PC + sext(imm)	jal x1, label	rd (5 bit); imm (20 bit)
JALR	Jump and Link Register	I	t = PC + 4; PC = (R[rs1] + sext(imm)) & ∼1; R[rd] = t	jalr x1, x5, 4	rd (5 bit); rs1 (5 bit); imm (12 bit)
BEQ	Branch if Equal	B	if (R[rs1] == R[rs2]) PC = PC + sext(imm)	beq x5, x6, imm	rd (5 bit); rs1 (5 bit); imm (12 bit)
BNE	Branch if Not Equal	B	if (R[rs1] != R[rs2]) PC = PC + sext(imm)	bne x5, x6, imm	rd (5 bit); rs1 (5 bit); imm (12 bit)
BLT	Branch if Less Than (Signed)	B	if (R[rs1] <s R[rs2]) PC = PC + sext(imm)	blt x5, x6, imm	rd (5 bit); rs1 (5 bit); imm (12 bit)
BGE	Branch if Greater or Equal (Signed)	B	if (R[rs1] >=s R[rs2]) PC = PC + sext(imm)	bge x5, x6, imm	rd (5 bit); rs1 (5 bit); imm (12 bit)
BLTU	Branch if Less Than (Unsigned)	B	if (R[rs1] <u R[rs2]) PC = PC + sext(imm)	bltu x5, x6, imm	rd (5 bit); rs1 (5 bit); imm (12 bit)
BGEU	Branch if Greater or Equal (Unsigned)	B	if (R[rs1] >=u R[rs2]) PC = PC + sext(imm)	bgeu x5, x6, imm	rd (5 bit); rs1 (5 bit); imm (12 bit)
LB	Load Byte	I	R[rd] = sext(Mem(R[rs1] + sext(imm), 1))	lb x5, imm(x6)	rd (5 bit); rs1 (5 bit); imm (12 bit)
LH	Load Halfword	I	R[rd] = sext(Mem(R[rs1] + sext(imm), 2))	lh x5, imm(x6)	rd (5 bit); rs1 (5 bit); imm (12 bit)
LW	Load Word	I	R[rd] = Mem(R[rs1] + sext(imm), 4)	lw x5, imm(x6)	rd (5 bit); rs1 (5 bit); imm (12 bit)
LBU	Load Byte Unsigned	I	R[rd] = zext(Mem(R[rs1] + sext(imm), 1))	lbu x5, imm(x6)	rd (5 bit); rs1 (5 bit); imm (12 bit)
LHU	Load Halfword Unsigned	I	R[rd] = zext(Mem(R[rs1] + sext(imm), 2))	lhu x5, imm(x6)	rd (5 bit); rs1 (5 bit); imm (12 bit)
SB	Store Byte	S	Mem(R[rs1] + sext(imm), 1) = R[rs2][7:0]	sb x5, imm(x6)	rd (5 bit); rs1 (5 bit); imm (12 bit)
SH	Store Halfword	S	Mem(R[rs1] + sext(imm), 2) = R[rs2][15:0]	sh x5, imm(x6)	rd (5 bit); rs1 (5 bit); imm (12 bit)
SW	Store Word	S	Mem(R[rs1] + sext(imm), 4) = R[rs2]	sw x5, imm(x6)	rd (5 bit); rs1 (5 bit); imm (12 bit)
ADDI	Add Immediate	I	R[rd] = R[rs1] + sext(imm)	addi x5, x6, imm	rd (5 bit); rs1 (5 bit); imm (12 bit)
SLTI	Set Less Than Immediate	I	R[rd] = (R[rs1] <s sext(imm)) ? 1 : 0	slti x5, x6, imm	rd (5 bit); rs1 (5 bit); imm (12 bit)
SLTIU	Set Less Than Immediate Unsigned	I	R[rd] = (R[rs1] <u zext(imm)) ? 1 : 0	sltiu x5, x6, imm	rd (5 bit); rs1 (5 bit); imm (12 bit)
XORI	XOR Immediate	I	R[rd] = R[rs1] ^sext(imm)	xori x5, x6, imm	rd (5 bit); rs1 (5 bit); imm (12 bit)
ORI	OR Immediate	I	R[rd] = R[rs1] — sext(imm)	ori x5, x6, imm	rd (5 bit); rs1 (5 bit); imm (12 bit)
ANDI	AND Immediate	I	R[rd] = R[rs1] & sext(imm)	andi x5, x6, imm	rd (5 bit); rs1 (5 bit); imm (12 bit)
SLLI	Shift Left Logical Immediate	I	R[rd] = R[rs1] <<imm[4:0]	slli x5, x6, imm	rd (5 bit); rs1 (5 bit); imm (12 bit)
SRLI	Shift Right Logical Immediate	I	R[rd] = R[rs1] >>u imm[4:0]	srli x5, x6, imm	rd (5 bit); rs1 (5 bit); imm (12 bit)
SRAI	Shift Right Arithmetic Immediate	I	R[rd] = R[rs1] >>s imm[4:0]	srai x5, x6, imm	rd (5 bit); rs1 (5 bit); imm (12 bit)
ADD	Add	R	R[rd] = R[rs1] + R[rs2]	add x5, x6, x7	rd (5 bit); rs1 (5 bit); rs2 (5 bit)
SUB	Subtract	R	R[rd] = R[rs1] - R[rs2]	sub x5, x6, x7	rd (5 bit); rs1 (5 bit); rs2 (5 bit)
SLL	Shift Left Logical	R	R[rd] = R[rs1] <<R[rs2][4:0]	sll x5, x6, x7	rd (5 bit); rs1 (5 bit); rs2 (5 bit)
SLT	Set Less Than (Signed)	R	R[rd] = (R[rs1] <s R[rs2]) ? 1 : 0	slt x5, x6, x7	rd (5 bit); rs1 (5 bit); rs2 (5 bit)
SLTU	Set Less Than Unsigned	R	R[rd] = (R[rs1] <u R[rs2]) ? 1 : 0	sltu x5, x6, x7	rd (5 bit); rs1 (5 bit); rs2 (5 bit)
SRL	Shift Right Logical	R	R[rd] = R[rs1] >>u R[rs2][4:0]	srl x5, x6, x7	rd (5 bit); rs1 (5 bit); rs2 (5 bit)
SRA	Shift Right Arithmetic	R	R[rd] = R[rs1] >>s R[rs2][4:0]	sra x5, x6, x7	rd (5 bit); rs1 (5 bit); rs2 (5 bit)
OR	OR	R	R[rd] = R[rs1] — R[rs2]	or x5, x6, x7	rd (5 bit); rs1 (5 bit); rs2 (5 bit)
AND	AND	R	R[rd] = R[rs1] & R[rs2]	and x5, x6, x7	rd (5 bit); rs1 (5 bit); rs2 (5 bit)
XOR	XOR	R	R[rd] = R[rs1] ^R[rs2]	xor x5, x6, x7	rd (5 bit); rs1 (5 bit); rs2 (5 bit)
FENCE	Fence Memory/I-O	I	Synchronizes memory and I/O	fence	-
ECALL	Environment Call	I	Transfers control to the O.S.	ecall	-
EBREAK	Environment Breakpoint	I	Transfers control to the debugger	ebreak	-

*sext(sign-extension)

Table 2.2 explicitly defines the structural format, operand syntax, and exact logical-mathematical execution behavior expressed in formal hardware

pseudocode, for each standard computational and control flow instruction, according to the official RISC-V specification.

2.3 Design of the Hardware *Debug-Module*

This subsection describes the *Debug-Module*, a fundamental macro-block of the Microprocessor designed to monitor the circuit during execution and verify its correct logical and functional behavior.

The debugger interfaces serially with external hardware to enable firmware execution control and real-time monitoring of individual internal components. Specifically, it features direct memory access acting as a Direct Memory Access (DMA) controller to the system storage structures (*Data-Memory*, *Code-Memory*, and the *Local Register* file), allowing the external host to both initialize and read their contents.

External communication is performed via a synchronous serial interface that stores incoming bit-streams into an internal 40-bit shift register.

Input serial data transmission is carried by the *serial_i* signal, which sequentially loads the shift register, while output serial transmission is handled by the *serial_o* signal.

Both serial pathways utilize two dedicated signaling lines to govern the communication protocol:

- ***serial_rdy_o*:** An output bit-flag signal that asserts when the shift register has been completely filled and is ready for internal processing, or when parallel data has been serialized and is ready for external output transmission.

- ***clock_serial*:** An input clock signal that dictates the data transmission and reception frequency. This clock is supplied by the external host system at an operational frequency of 100 MHz.

Table 2.3: Structural field mapping and bit allocation of the debug shift register.

Data [39:8]	**Address [7:3]**	**Memory Type [2:1]**	**Mode [0]**
If Mode == 1 32 bit word If Mode == 0 don't care, bits are replaced with the queried value	Range 0 to 31	00: *Data-Memory* 01: *Code-Memory* 10: *Local Register* 11: I/O	0: Read 1: Write

Table 2.3 defines the topology of the 40-bit packet received by the *Debug-Module* via the serial interface. The 40-bit packet is logically partitioned into four distinct functional fields:

- **Mode [0]:** A 1-bit field that specifies the operation type, defining either a memory write command (logical high, *1*) or a memory read command (logical low, *0*).

- **Address [7:3]:** A 5-bit field that specifies the address pointer of the target memory location scheduled for a read or write operation.
- **Data [39:8]:** A 32-bit field containing the data payload. When a write command is issued, this field holds the 32-bit word to be stored into the memory cell defined by the Address field.
- **Memory Type [2:1]:** A 2-bit field that selects the specific target memory subsystem involved in the operation.

Additionally, the *Debug-Module* drives a core enable signal (*core_en*, provided externally as an input to the Microprocessor) that enables (logical high, *1*) or disables (logical low, *0*) the execution of the Microprocessor core. Through this control signal, the debugger can freeze the logical state of the circuit, allowing step-by-step investigation of its behavior during every intermediate cycle of firmware execution.

During the system initialization phase, the debugger disables the *core_en* signal to halt the Microprocessor and initializes the memory arrays via the input serial interface. Subsequently, it asserts the core execution, allowing the core to process the firmware binary loaded into the *Code-Memory*. The debugger can halt execution at any time by disabling the core to read back the internal memory states.

The internal memory subsystems represent the ultimate logical endpoints of any hardware execution. By verifying their contents at each intermediate cycle, it is possible to identify the exact location of faults or timing bugs that would otherwise remain hidden if only the final output of the executed firmware were observed.

In conclusion, the *Debug-Module* is an essential block that provides a robust external interface and complete observability over the Microprocessor architecture. The debugger has been specifically engineered to ensure full equivalence between post-layout simulations and experimental laboratory measurements. This granular control allows the identical verification procedure to be deployed across both simulation frameworks and physical silicon characterization, which is paramount for the robust verification and characterization of complex integrated systems like general-purpose Microprocessors.

2.4 Design and Addressing Topology of *Code-Memory* and *Data-Memory*

This subsection describes the memory elements of the Microprocessor, namely the *Code-Memory* and the *Data-Memory*.

The *Code-Memory*, initialized by the external hardware *Debug-Module*, stores the instruction set (firmware) scheduled for execution. In this specific

implementation, its internal structure consists of 32 rows, each featuring a 32-bit width (columns). Consequently, the total allocation space comprises 32 words, equivalent to 128 bytes.

Conversely, the *Data-Memory*, also initialized by the hardware debugger, holds the datasets and variables previously processed by the core that are required to perform the operations dictated by the active firmware. Its hardware architecture is organized into 128 rows, each with an 8-bit width, yielding a total capacity of 128 bytes.

Table 2.4: Memory allocation space, register addresses and read/write permissions.

ADDRESS RANGE	MEMORY SIZE	DESCRIPTION	Read Core	Write Core	Read Debug	Write Debug
0x 000 – 0x 07F	128 bytes	Data Memory	Y	Y	Y	Y
0x 080 – 0x 0FF	128 bytes	Instructions Memory	Y	N	Y	Y
0x 100 – 0x 11F	128 bytes	Internal Register	Y	Y	Y	Y
0x 120 – 0x17F	-	Unused Addresses	N	N	N	N
0x 180 – 0x 183	4 bytes	Program Counter	N	N	Y	Y
0x 184 – 0x 187	4 bytes	Stop Core on RET	N	N	Y	Y
0x 188 – 0x 1EF	-	Unused Addresses	N	N	N	N
0x 1F0 – 0x 1F3	4 bytes	Serial Input	Y	Y	Y	Y
0x 1F4 – 0x 1F7	4 bytes	Data Memory	Y	Y	Y	Y
0x 1F8 – 0x 1FB	4 bytes	Serial Output	Y	Y	Y	Y
0x 1FC	1 byte	Output Pins	Y	Y	Y	Y
0x 1FD – 0x 1FF	3 bytes	Data Memory	Y	Y	Y	Y

Table 2.4 illustrates the internal structure, address mapping of the memory registers, and the access permissions (read and write capabilities) granted to the other macro-blocks within the Microprocessor architecture.

The structural asymmetry between the two subsystems is intrinsically dictated by their primary functional applications. The *Code-Memory* is accessed during the instruction *FETCH* phase to determine the subsequent instruction to be executed. Because the RV32I standard represents every instruction as a 32-bit word, with each instruction format assigning distinct logical meanings to these 32 bits, the hardware topology of the *Code-Memory* is optimally implemented as a register array structured in full words.

On the other hand, the *Data-Memory* is accessed during the *EXECUTE* phase depending on the specific instruction under processing. Certain operations manipulate only fractions of the standard 32-bit data representation. Specifically, these instructions can directly address *half-words* (16 bits) or single *bytes* (8 bits), which represent sub-portions of the 32-bit datapath established by the RV32I architecture. For this reason, the *Data-Memory* hardware is organized with byte-granularity. This approach minimizes the active memory segment triggered during execution, thereby significantly reducing power consumption. Furthermore, byte-level addressability enhances architectural flexibility, providing a more efficient mapping for typical software-hardware memory interactions.

Memory subsystems constitute essential endpoints that bridge successive hardware operations. By monitoring the internal memory states at each

intermediate execution cycle of the firmware, the correct logical and functional behavior of the Microprocessor can be accurately validated. Relying solely on the final output of a firmware program may obscure transient faults or masking errors that do not directly alter the end result. This deep logical inspection of the memory states during execution is managed by the hardware debugger, which can halt the core execution and read the memory contents at any clock cycle.

The storage capacities of these memories were deliberately kept small to constrain the structural complexity of the core's internal control logic. It is important to note that the main memory arrays are external to the standard RISC-V specification and can be engineered in various configurations including off-chip architectures integrated via advanced caching techniques. Therefore, scaling their dimensions does not alter the internal data-path structure of the core. However, for aerospace applications, memory subsystems require a significantly higher level of complexity due to their extreme sensitivity to radiation-induced degradation.

At the conclusion of this dissertation, the research and development carried out to enhance the robustness of these memory structures against aerospace radiation damage will be detailed. Specifically, Chapter 7 will present the integration of a Radiation-Tolerant-By-Design (RTBD) Static Random-Access Memory (SRAM) architecture that avoids traditional Triple Modular Redundancy (TMR). The design and physical implementation of three distinct SRAM topologies developed via an SRAM compiler, leveraging the digital standard-cell libraries of the commercial TSMC 28 nm CMOS Bulk process node will be thoroughly characterized.

2.5 Architectural Design and Pinout Specification of the *I/O-Module*

For Microprocessors, the ability to interface with the external environment through diverse communication protocols is a fundamental requirement. This subsection describes the input and output signals, alongside the dedicated interface macro-block, engineered for the proposed Microprocessor.

Table 2.5 details the comprehensive input/output pinout specification of the Microprocessor. External communication is managed and routed primarily through two dedicated functional macro-blocks: the *Debug-Module* and the *I/O-Module*. The signaling lines associated with the hardware debugger (as detailed in Section 2.3) are dedicated to the low-level control, configuration, and monitoring of the Microprocessor state.

In addition to the serial debug lines previously discussed, the debugger utilizes the following input and output signals to govern the core execution:

Table 2.5: Microprocessor top-level pinout and signal description.

Name	Direction	Bit Width	Description
en	Input	1	Master enable signal for the *Debug-Module*.
core_en	Input	1	Enable/freeze control signal for the *RV32I-Core*.
clk	Input	1	Primary system clock signal.
nrst	Input	1	Active-low, synchronous master reset signal.
external_interrupt_top_i	Input	1	Asynchronous external interrupt request line.
is_ret_top_o	Output	1	Firmware execution complete and debugger return flag.
clk_serial	Input	1	Clock signal for the debug serial communication.
serial_i	Input	1	Serial data input for the debugger shift register.
serial_o	Output	1	Serial data output from the debugger interface.
serial_rdy_o	Output	1	Debugger shift register ready or transmission status flag.
gpo_o	Output	8	General-Purpose Output parallel port.
gp_clk_serial_in_i	Input	1	Serial clock for the general-purpose input interface.
gp_data_wr_i	Input	1	Input shift register valid data flag.
gp_serial_in_i	Input	1	General-purpose serial data input line.
gp_clk_serial_out_i	Input	1	Serial clock for the general-purpose output interface.
gp_serial_out_o	Output	1	General-purpose serial data output line.
data_rd_i	Input	1	Output shift register serialization trigger signal.

- **en:** (Input) A master enable signal that activates the internal logic of the *Debug-Module*.
- **core_en:** (Input) A control enable signal that activates or freezes the operation of the *RV32I-Core*.
- **clk:** (Input) The primary system clock signal driving the Microprocessor.
- **nrst:** (Input) An active-low, synchronous master reset signal that initializes the Microprocessor state on the rising edge of the system clock.
- **external_interrupt_top_i:** (Input) A dedicated asynchronous interrupt request line that halts standard core execution to handle external exception routines.
- **is_ret_top_o:** (Output) A handshaking status flag indicating that the core has completed the execution of the active firmware routine and is returning control to the host debugger.

The *I/O-Module* macro-block handles general-purpose external communication by providing custom programmable input and output interfaces. Specifically, the peripheral subsystem integrates three distinct hardware interfaces:

- **1 Serial Input**
- **1 Serial Output**
- **8 General-Purpose Output (GPO) ports**

The *Serial Input* interface manages incoming serial bit-streams directed to the Microprocessor datapath. This interface is highly flexible and can be adapted to various system integration scenarios, such as receiving digital samples from an external Analog-to-Digital Converter (ADC). The incoming

serial stream is sequentially captured and parallelized into an internal 32-bit shift register.

The *Serial Input* interface utilizes three dedicated signaling lines to govern the communication protocol:

- **gp_clk_serial_in_i:** (Input, up to 100 MHz) A dedicated serial clock line that dictates the operating frequency of the input interface, allowing the physical layer to adapt to various external data rates.
- **gp_data_wr_i:** (Input) A status indicator flag for the input shift register that asserts when the parallelized data is valid and ready to be read by the core.
- **gp_serial_in_i:** (Input) The physical serial data input line.

The *Serial Output* interface manages outgoing data serialization from the Microprocessor to external devices. Similar to the input channel, this block can be interfaced with a variety of external components, such as a Digital-to-Analog Converter (DAC). The outbound data word is loaded into a 32-bit shift register before serialization.

The *Serial Output* engine utilizes three dedicated signaling lines to synchronize data transfers:

- **gp_clk_serial_out_i:** (Input, up to 100 MHz) A dedicated serial clock line that dictates the transmission frequency, enabling compliance with different external peripheral speed specifications.
- **gp_serial_out_o:** (Output) The physical serial data output line.
- **data_rd_i:** (Input) A handshake status trigger that signals the output shift register to initiate the serialization and transmission of a new data word.

The *8 GPO* ports provide direct-mapped parallel output pins. These pins are hardwired directly to the final byte location of the *Data-Memory* space. Consequently, the logic state of each individual GPO pin is structurally dependent on and instantaneously updated by the specific data written to that memory address during firmware execution.

2.6 FPGA-Based Prototyping and Hardware Verification on Xilinx Spartan-7

This subsection presents the implementation of the designed Microprocessor on the Xilinx Spartan-7 programmable fabric, integrated within the Digilent Cmod S7 development board framework.

The integration and hardware validation processes typically require extensive physical engineering, which significantly prolongs the architectural verification phase. Furthermore, as circuit complexity scales, a more rigorous and granular analysis of the synthesized netlist becomes mandatory. To accelerate the verification turnaround time of the designed circuit, an intermediate prototyping stage using an FPGA is introduced.

FPGAs enable preliminary evaluation on programmable hardware, eliminating the levels of abstraction inherent to pure behavioral or functional simulations. Consequently, the resulting hardware emulations are more realistic and closely match the physical behavior of the final silicon implementation.

The deployment workflow on the Spartan-7 FPGA architecture follows six fundamental steps:

1. **Source File Import:** The Hardware Description Language (HDL) code is loaded into the Integrated Development Environment (IDE), specifically Xilinx Vivado.

2. **Compilation and Syntax Checking:** The code-base undergoes compilation to perform initial syntax verification, semantic checks, and linting.

3. **Logic Synthesis:** The behavioral HDL descriptions are translated (synthesized) into a netlist of primitive hardware components, primarily Look-Up Table (LUT)s and registers, mapped specifically onto the target Spartan-7 FPGA resources.

4. **Implementation and Bitstream Generation:** The synthesized netlist undergoes placement and routing to optimize interconnect timing inside the Cmod S7 fabric, followed by the generation of the configuration bitstream file.

5. **Hardware Programming:** The generated bitstream is downloaded to the Cmod S7 board via its onboard USB-JTAG bridge, initializing the programmable logic cells.

6. **Execution and Runtime Validation:** The FPGA hardware is fully configured and ready to execute the synthesized Microprocessor circuit at the target clock frequency.

The reconfigurable fabric of the Xilinx Spartan-7 is structurally organized as an array of logic blocks containing 6-input LUTs as their core primitive elements. These LUTs incorporate the fundamental digital structures (such as combinational logic primitives, multiplexers, and D-type Flip-Flops) required to implement arbitrary digital networks. When the Vivado synthesis engine defines the structural topology of the circuit, it programs the boolean truth tables of each individual LUT and configures the programmable routing matrices to establish the interconnects.

By leveraging the Digilent Cmod S7 for prototyping, an in-depth architectural study can be conducted without executing a full application-specific integrated circuit (Application-Specific Integrated Circuits (ASIC)) digital design flow. Synthesizing the design into primitive digital components (LUTs) allows for the early identification of logical exceptions, as well as a preliminary evaluation of timing margins and static/dynamic power dissipation within the 28nm FPGA technology.

The Microprocessor is architected to operate at a target clock frequency of 100 MHz. This clock constraint is strictly bound by the system's critical path. The critical path represents the sequential path comprising combinational logic cells and interconnect routing wires that exhibits the maximum propagation delay between any two synchronous registers. The total propagation delay along this pathway dictates the maximum theoretical operating frequency ($f_{\max}$) at which the circuit can operate without violating *Setup* and *Hold* timing constraints.

FPGA prototyping enables an empirical investigation and validation of this critical path. Given the strict timing constraints associated with the 100 MHz target clock, if the Static Timing Analysis (STA) engine within Vivado fails to achieve timing closure (i.e., resulting in a negative worst-case slack), it indicates that the hardware cannot guarantee correct deterministic operation under all operating conditions at the target frequency.

Through the development environment's advanced reporting tools (Vivado Timing Reporter), the post-synthesis and post-implementation schematics can be thoroughly analyzed. This structural observability allows designers to isolate the critical timing bottlenecks and optimize the underlying Verilog code accordingly. This optimization process is paramount for complex digital systems like Microprocessors, where predicting the exact physical hardware translation of high-level HDL descriptions during the initial design phase is highly challenging.

Table 2.6: Post-synthesis and post-implementation LUT resource utilization summary.

Resource	**Post-Synthesis**		**Post-Implementation**		**Available**
	Utilization	%	**Utilization**	%	
LUT	669	4.58	2813	19.27	14600
FF	675	2.31	2015	6.90	29200
BRAM	4	8.89	4.50	10.00	45
IO	28	18.67	28	16.67	150
BUFG	2	6.25	4	12.50	32

Table 2.6 and Figure 2.5 summarize the primary implementation data of the Microprocessor deployed on the FPGA fabric. Specifically, the table

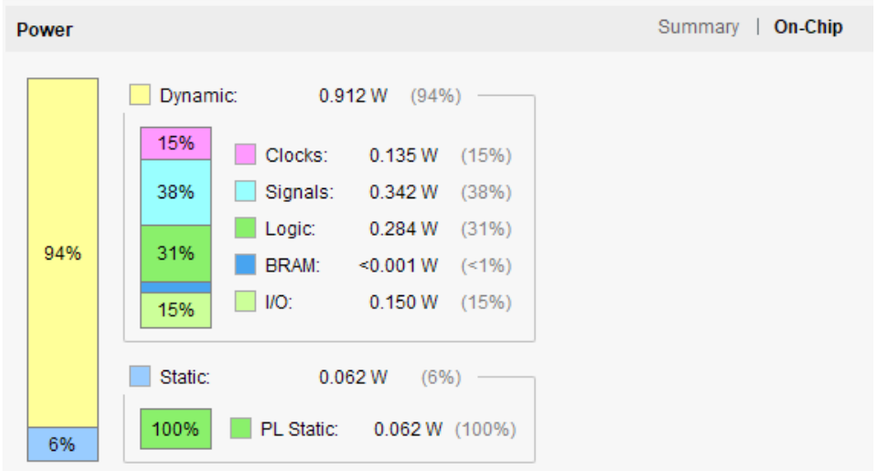

Figure 2.5: Static and dynamic power estimation generated by the Vivado power analysis tool.

compares the LUT resource utilization metrics obtained post-synthesis and post-implementation, while the Figure details the power dissipation profile, isolating the individual power consumption contributions of the major FPGA hardware primitives and the synthesized Microprocessor macro-blocks.

Chapter 3

Digital Design Flow and ASIC Implementation

This chapter presents the digital design flow implemented to transition the previously discussed Microprocessor architecture into a physical hardware implementation. The sequence of operations executed during the logic *Synthesis* and *Place&Route* phases will be detailed, with particular emphasis on the specific constraints, design methodologies, and optimization parameters applied to enhance the robustness and reliability of the integrated circuit for aerospace applications.

To ensure high robustness, the circuit must be thoroughly verified across multiple operational boundary conditions, specifically leveraging Process-Voltage-Temperature (PVT) corners and Multi-Corner Multi-Mode (MCMM) analysis frameworks, which are maintained throughout the entire digital implementation flow. Furthermore, to maximize reliability under extreme environmental stress, the digital flow must be tailored to minimize the peak *IR-Drop* (voltage degradation) within the *Power-Grid*, especially when accounting for Single-Event Effects (SEE) induced by aerospace radiation. In a physical environment, the transient current pulses generated by an SEE, when superimposed on a severe local *IR-Drop*, can dangerously degrade noise margins and alter the internal logical state of the Microprocessor, leading to catastrophic functional failures. In addition, deep focus will be placed on the methodologies adopted to mitigate the leakage current contributions inherently associated with the selected Bulk Complementary Metal-Oxide-Semiconductor (CMOS) technology node.

Figure 3.1 illustrates the complete design flow developed to translate the behavioral Hardware Description Language (HDL) description presented in the preceding chapters into a physical Application-Specific Integrated Circuits (ASIC). The flowchart outlines the intermediate execution stages required by the physical design flow. Additionally, it highlights the exact steps within the workflow where specific layout and optimization modifications were injected to harden the circuit and mitigate sub-threshold leakage currents.

The digital implementation flow converts high-level, behavioral HDL source

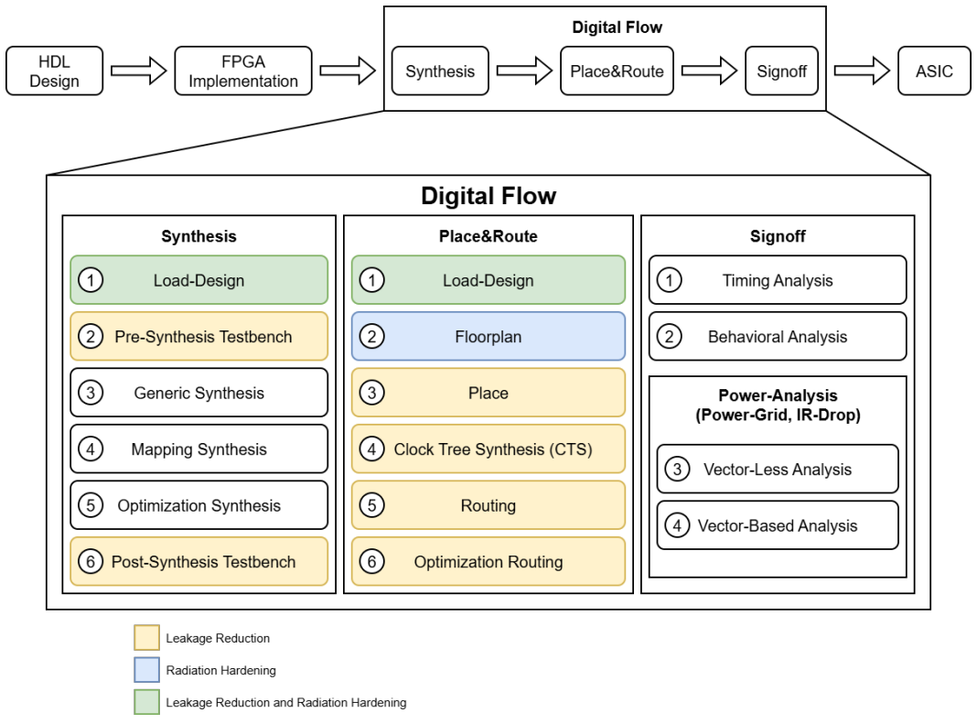

Figure 3.1: Block diagram of the top-down digital flow stage to leakage reduction and radiation hardening.

code into an equivalent, physically realizable hardware netlist. This optimization process is primarily split into two main macro-phases: logic *Synthesis* and *Place&Route*. These phases are systematically followed by a rigorous *Signoff* verification stage, where the finalized integrated circuit is validated through post-layout functional simulations, Static Timing Analysis (STA) to satisfy timing constraints, functional-logic behavior and extensive power analysis including both average power estimation and *Power-Grid IR-Drop* verification.

Both the synthesis and implementation engines rely heavily on the technology standard-cell libraries provided by the foundry, which in this work is Taiwan Semiconductor Manufacturing Company (TSMC). These libraries deliver a comprehensive set of primitive and complex logic gates, highly optimized and characterized by the foundry for digital logic implementation.

The logic *Synthesis* phase ingests the high-level HDL description (written in Verilog) and, through logic minimization and technology mapping algorithms, translates the behavioral constructs into an optimized gate-level netlist using the cells available in the technology library. Conversely, the *Place&Route* phase takes this gate-level netlist as an input to perform the physical layout generation. Specifically, during the placement stage, the physical positions of the standard cells are determined within the core area, while during the routing stage, the physical metal interconnects are drawn to establish all electrical connections.

By analogy with traditional analog design, the logic *Synthesis* phase can be conceptually mapped to schematic design, whereas the *Place&Route* phase

represents the generation of the physical layout.

3.1 Standard-Cell Library Characterization and Multi-Corner Multi-Mode (MCMM) Framework

This subsection provides an exhaustive analysis of the digital standard-cell libraries, elucidating their organizational taxonomies and the heterogeneous file formats that characterize individual cells across distinct abstraction levels. Specifically, this work leverages the commercial TSMC 28 nm CMOS HPC+ Bulk technology node libraries to physically realize the proposed Microprocessor architecture into an ASIC.

The TSMC standard-cell library distribution is structured via a multi-level hierarchical directory framework. Navigating down this hierarchy progressively refines the geometric and electrical specifications of the target cells.

The primary hierarchical level categorizes the standard cells based on their cell height, quantified by the number of routing tracks. This metric directly defines the number of metal routing tracks available over the cell area:

- **7-track (7T):** High-density, low-power footprint.
- **9-track (9T) [Selected]:** Standard balanced profile optimizing both performance and area efficiency.
- **12-track (12T):** High-performance footprint for critical path acceleration.

The secondary hierarchical layer differentiates the standard cells according to their minimum nominal transistor gate length (L_{min}):

- **tcbn28hpcplusbwp30p140-set:** 30 nm minimum channel length.
- **tcbn28hpcplusbwp35p140-set [Selected]:** 35 nm minimum channel length.
- **tcbn28hpcplusbwp40p140-set:** 40 nm minimum channel length.

The tertiary hierarchical tier distinguishes the digital cells based on functional categories, primarily determined by their threshold voltage (V_{th}) profiles and structural sub-types:

- **tcbn28hpcplusbwp35p140_190a_FE:** Standard Threshold Voltage (SVT) variant [Selected].
- **tcbn28hpcplusbwp35p140cg_150a_FE:** Coarse-Grain cell structures.

- **tcbn28hpcplusbwp35p140hvt_190a_FE:** High Threshold Voltage (HVT) variant, optimized for low leakage [Selected].
- **tcbn28hpcplusbwp35p140lvt_190a_FE:** Low Threshold Voltage (LVT) variant, optimized for high speed [Selected].
- **tcbn28hpcplusbwp35p140opp_110c_FE:** Optimized Power (OPP) structures.
- **tcbn28hpcplusbwp35p140opphvt_110c_FE:** Optimized Power with High Threshold Voltage.
- **tcbn28hpcplusbwp35p140opplvt_110c_FE:** Optimized Power with Low Threshold Voltage.
- **tcbn28hpcplusbwp35p140uhvt_190a_FE:** Ultra-High Threshold Voltage (UHVT) variant.
- **tcbn28hpcplusbwp35p140ulvt_190a_FE:** Ultra-Low Threshold Voltage (ULVT) variant.

The quaternary hierarchical level contains the actual workspace views that characterize the chosen standard-cell flavor. These views are divided into three main operational directories:

- **Back-end:** Contain the physical implementation layouts and abstract geometries. The files contained herein (e.g., LEF, GDSII) are primarily ingested during the *Place&Route* and physical validation *Signoff* phases.
- **Documentation:** Provides data-sheets and lookup tables detailing raw cell parameters across distinct operating points.
- **Front-end:** Contains the logical functionality, timing lookup models, and power models (e.g., Liberty files .lib). These views are heavily utilized during logic *Synthesis* and power *Signoff*.

These views characterize cell behavior across varying Process-Voltage-Temperature (PVT) corners, defined by the following parametric ranges:

- **Temperature:** -40°C, 25°C, 125°C.
- **Process Corners:** SS (Slow-Slow), SF (Slow-Fast), FS (Fast-Slow), TT (Typical-Typical), FF (Fast-Fast).
- **Supply Voltage ($V_{\mathbf{DD}}$):** 0.81 V, 0.90 V, 0.99 V.

A meticulous evaluation of these standard-cell parameters is paramount to properly structure the Electronic Design Automation (EDA) tool execution scripts. During the initial physical design initialization, the precise subsets of

Table 3.1: Standard-cell selection and MCMM configuration for *Setup* and *Hold* timing analysis.

Parameter	**Value**
Track Number	9
Transistor Minimum Gate Length	35 nm
Threshold Voltage	STV, HVT, LVT
Digital Standard Cell	tcbn28hpcplusbwp35p140_190a_FE tcbn28hpcplusbwp35p140hvt_190a_FE tcbn28hpcplusbwp35p140lvt_190a_FE
MCMM *Setup* Analysis	SS, 0.81 V, 125 °C, Worst SS, 0.81 V, -40 °C, Worst TT, 0.9 V, 25 °C, Typical
MCMM *Hold* Analysis	TT, 0.9 V, 25 °C, Typical FF, 0.99V, 0 °C, Best FF, 0.99V, 0 °C, Worst FF, 0.99V, -40 °C, Best FF, 0.99V, -40 °C, Worst FF, 0.99V, 125 °C, Best FF, 0.99V, 125 °C, Worst SS, 0.81 V, -40 °C, Worst SS, 0.81 V, 125 °C, Worst

standard cells and PVT corners must be explicitly selected based on circuit constraints.

To safely integrate the Microprocessor, a structured Multi-Corner Multi-Mode (MCMM) validation framework is established. This framework binds distinct physical design modes with target operational corners during both active placement/routing and final *Signoff* analysis. Table 3.1 summarizes the selected MCMM configuration, explicitly defining which corners govern *Setup* and *Hold* constraints. The underlying configuration script simultaneously defines the clock timing constraints and maps the interconnected parasitic RC models (e.g., Best, Typical, Worst corners) derived from the Cadence Quantus extraction engine.

The analysis configurations target the most critical operational conditions to prevent timing violations. These checks verify that the layout strictly satisfies both the *Setup-Time* and *Hold-Time* margins required by the synchronous flip-flops relative to the system clock.

The *Setup-Time* defines the minimum temporal window before the clock signal's *Rising-Edge* during which the data input of a sequential element must remain perfectly stable to ensure deterministic latching.

Conversely, the *Hold-Time* specifies the mandatory duration for which the data input must remain unaltered immediately following the clock signal's *Rising-Edge*. Simultaneous compliance with both constraints is required to prevent metastability and guarantee proper logic execution.

The *Setup* and *Hold* verification passes sweep across the full MCMM matrix to validate timing margins under worst-case corners. For *Setup-Time* analysis,

the limiting case occurs at the slowest operating corner (SS, −40°C, 0.81 V), where low supply voltages, worst-case slow silicon processing, and low temperature maximize combinational logic propagation delays. This degradation in switching speed makes it harder for data signals to arrive at the destination flip-flop inputs before the setup window closes.

Conversely, for *Hold-Time* verification, the most critical scenario shifts to the fastest operating corner (FF, 125°C, 0.99 V), where elevated supply voltages, fast silicon processing, and high temperature minimize cell propagation delays. This acceleration in switching speed increases the risk that a fast data wave-front propagates through the combinational logic too quickly, overwriting the data latching at the destination register before the mandatory *Hold-Time* window has completely expired.

3.2 Logic *Synthesis* and Leakage Mitigation Methodologies

This subsection describes the logic *Synthesis* phase (Cadence Genus), delineating the technical stratagems and constraints deployed to strengthen the circuit against aerospace induced SEE and to systematically suppress the static leakage current component. Static power mitigation is particularly critical given the selection of a bulk CMOS process node, which exhibits a higher baseline leakage profile compared to alternative fully depleted architectures.

Logic *Synthesis* constitutes the initialization stage of the physical design implementation flow. This phase ingests the high-level HDL description of the system and translates it into an optimized gate-level netlist using the primitive and complex standard cells provided in the foundry's technology library.

The initial phase of the *Synthesis* workflow (*Load Design*) involves importing the hierarchical HDL code, selecting the target standard-cell libraries, and defining the specific operational corners established within the MCMM framework (Table 3.1). The synthesized netlist must satisfy all timing, power, and area constraints across every distinct scenario defined in the MCMM execution scripts. This rigid constraint mapping represents the fundamental prerequisite to guarantee deterministic Microprocessor operation under harsh aerospace operating conditions.

Simultaneously, the *Synthesis* engine configuration requires the definition of global optimization attributes, including the parametric weighting factors that govern static and dynamic power trade-offs.

A comprehensive analysis of the TSMC 28 nm library documentation revealed a nominal technology-inherent static-to-dynamic power ratio of approximately 0.1, indicating that static leakage currents account for 10 % of the total power budget under typical operating conditions. This nominal value of

0.1 was rigorously validated via transistor-level simulations (Cadence Spectre) executed on a statistical sample of logic gates, and it aligns with baseline benchmarks documented in literature [82–84].

To actively mitigate the leakage currents within the bulk technology framework, the EDA optimization weight for the static-to-dynamic power ratio was strategically configured to 0.2 during both the logic *Synthesis* and *Place&Route* execution flows.

To ensure reproducibility, this is governed by a specific cost-function parameter within the Cadence environment, namely the *opt_leakage_to_dynamic_ratio* variable (configured via *set_db* in Genus and *setOptMode* in Innovus).

The digital implementation profile is finely calibrated via the target static-to-dynamic power ratio across the entire design flow: a setting of 0.0 focuses exclusively on dynamic power minimization, 1.0 isolates leakage mitigation, while 0.5 balances both components equally.

Deviating excessively from the baseline ratio dictated by the technology node leads to sub-optimal EDA tool convergence throughout the physical *Synthesis* and *Place&Route* phases this reduces the optimization effort on the power component with the highest impact on the total energy budget, thereby degrading overall performance.

Since physical and electrical characterization identified a nominal technology ratio of 0.1, the target static-to-dynamic ratio was consistently constrained to 0.2 during the entire digital flow. This strategic shift aggressively optimizes leakage by forcing the tools to prioritize low-leakage cell selection across all optimization stages.

Table 3.2: CAD Ablation Study: Comparison of Unconstrained and Leakage-Mitigated Digital Flows.

Parameters	**Units**	**opt_leakage_to_dynamic_ratio**			
		0.2		**0.0**	
		Typical	**Worst**	**Typical**	**Worst**
Static Current Cunsumption	**[mA]**	0,011	0,017	0,17	0,305
Dynamic Current Consumpiton	**[mA]**	1,776	2,39	1,974	3,277
Static-to-Dynamic	**[%]**	0,61	0,71	8,64	9,32
Setup-Time (Worst, SS, 0.81 V, 125°C)	**[ps]**	720		2344	
Hold-Time (Worst, FF, 0.99 V, -40°C)	**[ps]**	86		92	
HVT Cells Number	**[-]**	14214		4002	
SVT Cells Number	**[-]**	18929		16675	
LVT Cells Number	**[-]**	78		8244	

Typical (TT, 0.9 V, 25 °C), Worst (FF, 0.99 V, 125 °C)

To further validate this parameter selection alongside the initial Spectre simulations, a Computer-Aided Design (CAD) ablation study (Table 3.2) compared a default unconstrained digital flow (0.0 leakage bias) against the proposed 0.2 bias. Current extractions utilized full-chip testbenches across Typical (25 °C) and Worst-case (125 °C) corners. As detailed in the table,

without the leakage bias, the EDA tool over-optimizes for speed by aggressively inserting fast but leaky LVT cells. This yields unnecessary timing margins and a static-to-dynamic ratio bounded only by generic global optimizations (<10 %). Conversely, the 0.2 bias acts as a targeted architectural constraint: the tool strategically trades this excess timing slack to heavily prioritize HVT cells. This mechanism successfully suppresses the worst-case static current down to 0.017 mA at 125 °C, securely maintaining timing closure while maximizing the core's energy efficiency.

Following the initialization of these *Synthesis* attributes, the Verilog source files are parsed and elaborated. The elaboration stage performs semantic, syntactic, and structural checks to ensure the codebase can be correctly processed by the *Synthesis* compilation algorithms.

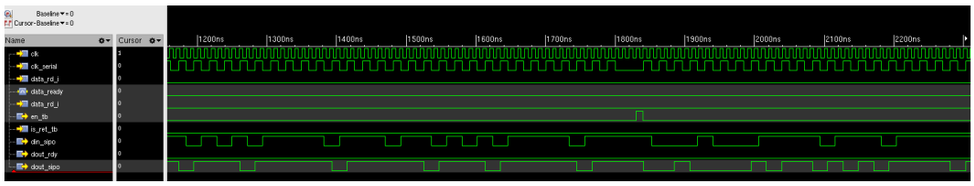

Figure 3.2: Pre-synthesis RTL simulation waveforms demonstrating ideal functional timing and zero-delay logic transitions.

Prior to gate-level mapping, a behavioral testbench is executed (*Pre-Synthesis Testbench*). As illustrated by the timing waveforms in Figure 3.2, this verification step is fully idealized and abstracts away all physical interconnect routing effects and cell propagation delays.

This functional simulation provides a golden reference baseline for subsequent post-*synthesis* equivalence verification, ensuring that the compiled gate-level netlist remains logically and functionally identical to the original Register-Transfer Level (RTL) description.

Furthermore, this initial behavioral simulation generates a Toggle Count Format (*.TCF*) file that quantifies the *Switching-Activity* of every internal node. It is vital that the verification testbench features high toggle coverage to comprehensively stimulate all internal logic blocks, which is highly challenging in complex architectures like Microprocessors. Insufficient testbench stimulus risks masking critical timing paths, *Setup/Hold* violations, or localized high-current vectors that induce severe *IR-Drop* during physical execution.

Once the baseline simulation pass is complete, the physical *Synthesis* flow executes across three discrete optimization stages:

- **Generic Synthesis:** The behavioral HDL constructs are elaborated into an intermediate, technology-independent netlist composed of abstract generic logic primitives.

- **Mapping Synthesis:** The generic logic structures are optimized and

mapped directly onto the physical standard cells available within the selected TSMC 28 nm technology libraries.

- **Optimization Synthesis:** The mapped gate-level netlist undergoes iterative optimization driven by the global timing, power, area, and design rule constraints defined during initialization.

Upon completion of the optimization passes, a comprehensive suite of design reports (encompassing timing slack, cell area, static/dynamic power, and design rule violations) is generated to evaluate netlist quality.

Subsequently, the gate-level netlist is re-verified using the same functional testbench used in pre-*Synthesis* to validate logical equivalence (*Post-Synthesis Testbench*). This step also generates a highly realistic post-*Synthesis* *.TCF* file capturing the updated *Switching-Activity*.

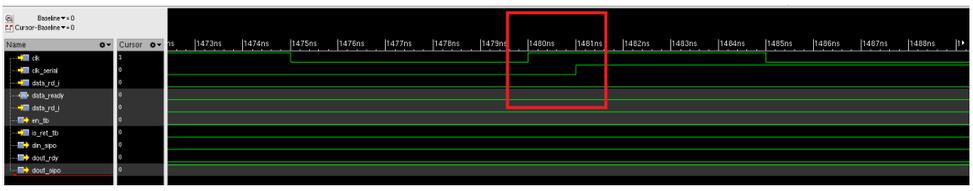

Figure 3.3: Post-synthesis gate-level simulation waveforms illustrating non-ideal cell propagation delays.

As demonstrated by the post-*Synthesis* waveforms in Figure 3.3, the signal transitions are no longer instantaneous; they exhibit non-ideal propagation delays and timing slacks derived directly from the look-up tables contained within the Front-end Liberty (.lib) models.

Figure 3.4 displays the initial macro-level cell floorplanning visualization obtained at the conclusion of the *synthesis* phase, which provides the baseline spatial boundaries required to calibrate subsequent *Place&Route* script parameters.

3.3 Physical Implementation: *Place&Route* Workflow

This subsection provides a detailed analysis of the *Place&Route* execution flow (Cadence Innovus), detailing the advanced design methodologies and layouts engineered to systematically mitigate leakage currents and harden the physical fabric against aerospace radiation environments, specifically targeting Single-Event Effects (SEE).

The *Place&Route* workflow ingests the gate-level structural netlist generated at the conclusion of the logic *Synthesis* phase and translates it into a physical ASIC layout mapped onto the selected technology node. This

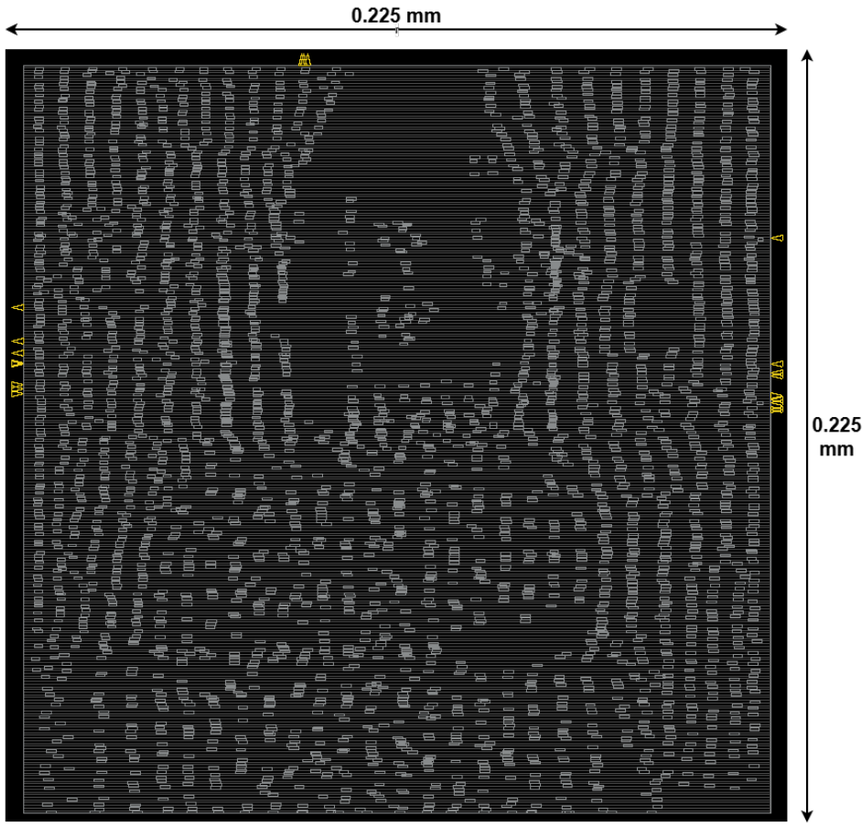

Figure 3.4: Post-*Synthesis* structural netlist showing digital-logic cell clustering.

macro-phase progresses through iterative optimization stages to achieve physical convergence, ensuring that the finalized layout complies with all logical, functional, static timing, and power dissipation constraints dictated by the MCMM infrastructure.

The initialization stage (*Load Design*) imports the gate-level netlist, binds the physical technology libraries, and activates the required MCMM operational analysis views. To ensure seamless continuity, the exact analysis views defined during the synthesis phase are preserved (Table 3.1). This multi-corner framework enhances the physical reliability of the integrated circuit by verifying timing closure across a parametric envelope that accommodates deviations exceeding 10 % for each physical variable (silicon process variation, supply voltage fluctuations, and temperature extremes).

During *Load Design*, the normalized static-to-dynamic power optimization weight is consistently constrained to 0.2, maintaining the leakage suppression strategy established during logic *Synthesis*.

Concurrently, baseline statistical *Switching-Activity* parameters are defined within the physical layout engine to drive early power and timing estimations. These include a global static *Toggle-Rate* configured to 0.5 (representing a 50 % node switching probability on every active clock edge) and a conservative switching margin booster factor set to 0.3 (*Increase Switch Consumption*). These internal parameters are augmented by importing the highly specific

Switching-Activity data enclosed within the Toggle Count Format (*.TCF*) file captured during post-*Synthesis* gate-level simulations. This dual-modeling approach provides a robust first-order approximation that guides the automated optimization engines toward valid convergence.

The static switching parameters are heavily leveraged during intermediate physical optimization steps to evaluate *Setup* and *Hold* timing slack before physical routing is finalized. Furthermore, these attributes act as the pessimistic inputs for statistical *Signoff* verification passes.

Given their impact on cell sizing and wire sizing, worst-case over-constrained values were intentionally selected relative to standard Microprocessor execution profiles to deliberately harden the physical design. Setting the nominal *Toggle-Rate* to 0.5 implies that half of the total standard-cell instances toggle simultaneously on every clock cycle, a condition that far exceeds typical instruction *Switching-Activity*.

Complementing this constraint, the 0.3 *Increase Switch Consumption* attribute acts as a guard band, forcing the optimization engines to calculate cell dynamic power with a pessimistic 30 % overhead compared to the nominal lookup tables enclosed in the foundry's Front-end Liberty (.lib) models.

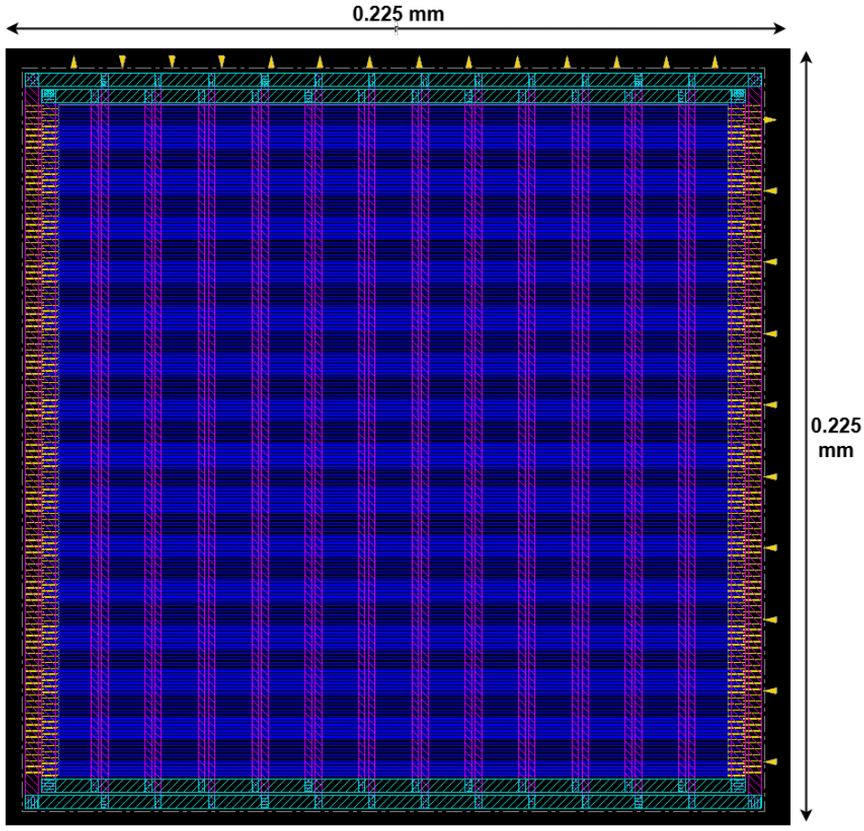

Figure 3.5: Core *Floorplan* layout configuration establishing the 0.225 mm square footprint and the dense vertical/horizontal *Power-Grid*.

The second execution stage (*Floorplan*) defines the core boundaries and constructs the physical *Power-Grid* upon which the standard-cell rows are structurally organized via followpins and power rings (Figure 3.5). The *Floorplan*

establishes the finalized silicon area footprint, in this case has been constrained to a square aspect ratio measuring exactly 0.225 mm × 0.225 mm.

To maximize physical robustness, a dense, low-impedance *Power-Grid* comprising concentric rings and vertical/horizontal stripes was engineered. This configuration minimizes the peak localized dynamic *IR-Drop*, safeguarding internal noise margins against transient current pulses induced by Single-Event Effects (SEE) in aerospace application.

The *Power-Grid* is strategically routed across the thick, top-level metal layers specifically leveraging Metal 7 (M7) for vertical paths and Metal 8 (M8) for horizontal paths due to their lower sheet resistance. Conversely, the internal row followpins are bound to Metal 1 (M1) to directly bias the power and ground rails of the standard-cell rails. The engineered grid topology integrates tightly spaced vertical stripes positioned at a uniform pitch of 14.165 μm with a width of 2 μm, bounded by concentric power and ground (V_{DD} and V_{SS}) rings measuring 4 μm in width.

The third stage is the physical placement (*Place*) phase, which distributes the standard-cell instances across the defined core rows. During this execution pass, the layout engine balances competing constraints including timing slack margins, localized routing density, cell area, and power metrics across the full MCMM corner matrix.

During cell placement, the Cadence Innovus engine executes an initial structural netlist restructuring. This step is necessary because logic *Synthesis* operates with a semi-abstracted model of physical interconnect parasitics. Consequently, a raw placement based strictly on logical connectivity can result in severe routing congestion or timing violations that cannot be resolved by subsequent optimization passes.

The greater the circuit complexity and the higher the target operating frequency, the more pronounced the divergence will be between the post-*Place&Route* implementation and the netlist generated during the logic *Synthesis* phase.

At the conclusion of the placement stage, the layout engine establishes a valid spatial distribution of cells, though it still relies on idealized models for wire routing and *Clock Tree* networks (Figure 3.6). At each intermediate checkpoint, the physical metrics are evaluated against target timing constraints. If the structural timing slacks or density metrics diverge beyond solvable optimization thresholds, the flow requires an execution rollback. Depending on the severity of the timing violations or critical path bottlenecks, the designer must adjust *Place&Route* parameters, execute a re-*Synthesis* pass with modified attributes, or modify the underlying high-level HDL source code.

To facilitate routing optimization and cell shifting, the maximum cell density constraint (*Max Density*) is capped at 0.85 (85 % area utilization). This intentional allocation of unplaced silicon area allows the optimization

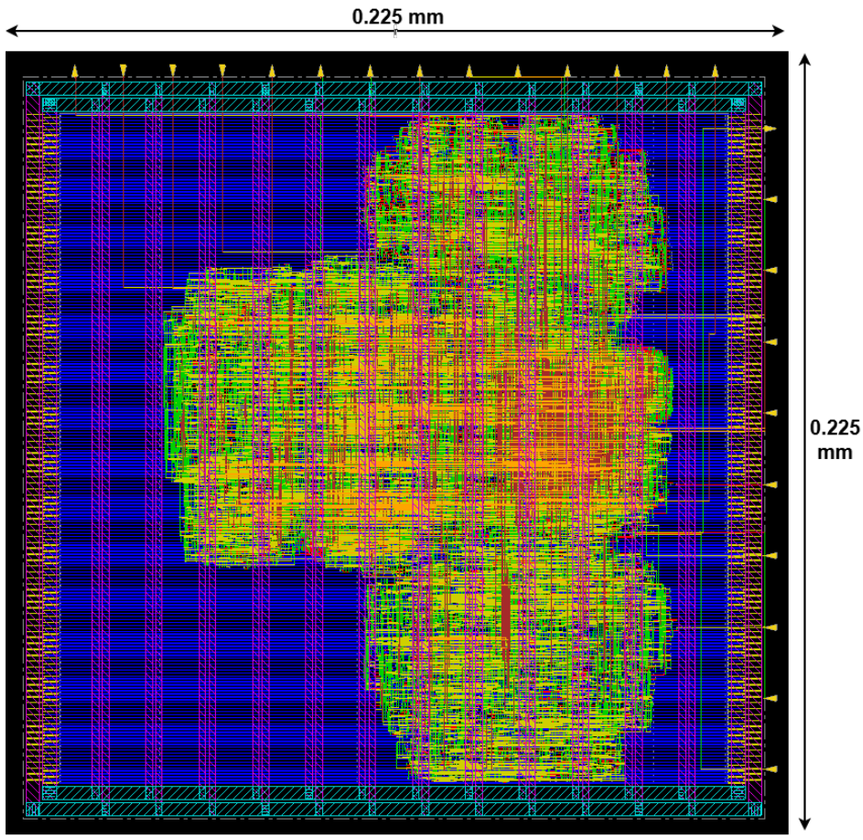

Figure 3.6: Standard-cell spatial distribution and legalized placement across core rows prior to *Clock Tree Synthesis*.

engine to inject auxiliary buffering or perform cell swapping without disrupting the legalized layout rows. Furthermore, a strict minimum instance spacing constraint (*Legalization Inst Gap*) is configured to a value of 2. This attribute forces a structural gap between adjacent cells equivalent to two minimum-width inverter cells within the target node fabric (28nm CMOS Bulk HPC+), ensuring available tracks for late-stage Engineering Change Orders (ECO) and layout fixing. (Note: To comply with the foundry Non-Disclosure Agreement (NDA), the absolute physical dimensions in micrometers for this minimum cell site cannot be explicitly disclosed).

Once a stable, legalized placement that satisfies timing, power, and *IR-Drop* targets across the MCMM matrix is achieved, the focus shifts to area minimization and layout tuning to recover performance margins.

The fourth stage is Clock Tree Synthesis (CTS), which constructs the specialized buffer networks required to distribute the clock signal uniformly across all sequential elements. In synchronous digital systems, the clock network governs the precise timing windows allocated for data propagation and latching.

A primary constraint during CTS is the minimization of global clock *Skew*, meaning that the propagation delay between the first cell and the last cell receiving the clock signal must be minimized to the greatest possible extent (restricted to a maximum threshold of 50 ps). Through iterative *Skew-*

balancing algorithms, the physical clock tree achieved a maximum worst-case *Skew* of just 18 ps under the most critical corner (SS, 0.81 V, 125°C). This variation represents a negligible 0.18% of the total 10 ns clock period dictated by the 100 MHz target operating frequency.

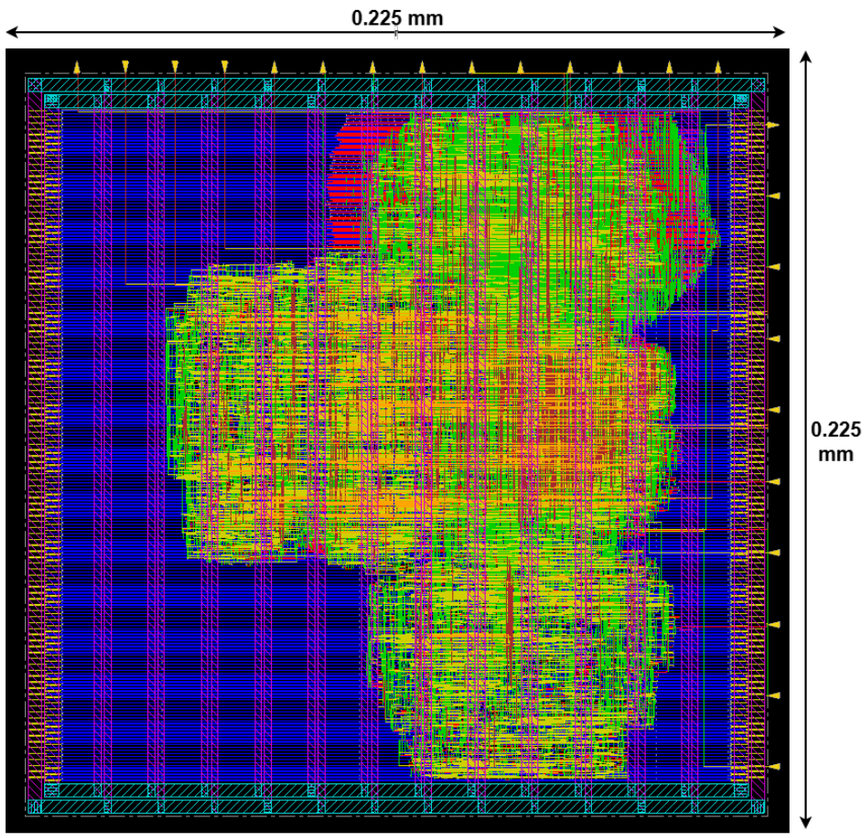

Figure 3.7: Physical layout view of the Microprocessor core after *Clock Tree Synthesis*.

The generated *Clock Tree* distribution topology is organized as a balanced tree structure to ensure equidistant temporal propagation to every sequential register (Figure 3.7). The clock network is physically segmented across three hierarchical routing tiers to minimize resistance and cross-talk:

- **Top:** Routed exclusively on the thick upper metal layers, M6 and M7.
- **Trunk:** Routed across intermediate metal layers, M5 and M6.
- **Leaf:** Routed on lower metal layers, M3 and M4, to interface directly with standard-cell clock pins.

During and immediately following CTS execution, the layout engine performs concurrent timing optimization, ensuring that the *Setup* and *Hold* times for all synchronous paths maintain positive slack against the baseline 100 ps library constraints. Given the 100 MHz operating frequency, these rigid timing targets provide a robust safety margin.

Following *Clock Tree Synthesis*, the fifth stage is the global and detail *Routing* phase. The routing engine replaces the abstract, straight-line logical

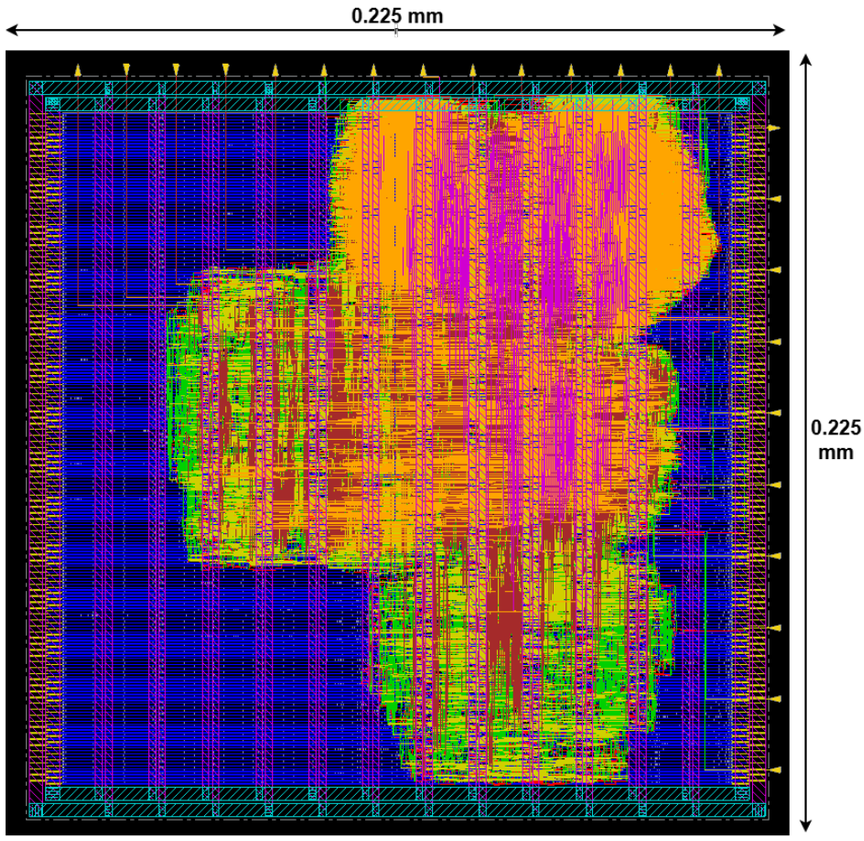

Figure 3.8: Post-*Routing* physical layout illustrating the fully materialized signal interconnect network and detailed geometric wire paths.

connections established during placement with actual physical metal interconnect wires, factoring in detailed *RC* parasitic models, lithographic manufacturing rules, and signal integrity constraints (Figure 3.8).

Upon completion of the detailed *Routing* pass, the design includes all physical wire geometries required for manufacturing. The layout engine then executes an optimization timing to resolve any late-stage *Setup* or *Hold* violations introduced by physical wire crossovers. Furthermore, integrability verification are performed:

- **Design Rule Checking (DRC):** Verifies that all layout geometries strictly comply with the geometrical, lithographic, and spacing rules mandated by the foundry's manufacturing process.
- **DRC-Antenna:** Evaluates the layout for process-induced antenna effects, ensuring that large metal wire areas connected to gate oxides do not collect charge during fabrication, which could rupture thin dielectrics or permanently alter transistor thresholds.
- **DRC-Density:** Validates that the global and local metal density percentages satisfy mechanical planarization thresholds, preventing structural deformation or mechanical failure of the *Die* during wafer fabrication.

If isolated DRC violations are identified, automated and manual layout

cleaning routines are executed, leveraging the structural gaps reserved during the placement phase.

When a *Place&Route* flow is properly configured with foundry technology files (e.g., LEF and tech-LEF models), large-scale DRC errors are naturally suppressed since manufacturing rules are actively enforced during wire drawing. Consequently, a excessive volume of DRC violations typically points to a fundamental flaw in *Floorplan* initialization, such as over-constraining the core area during the *Floorplan* stage, which forces the routing engine to generate illegal or un-routable wire geometries.

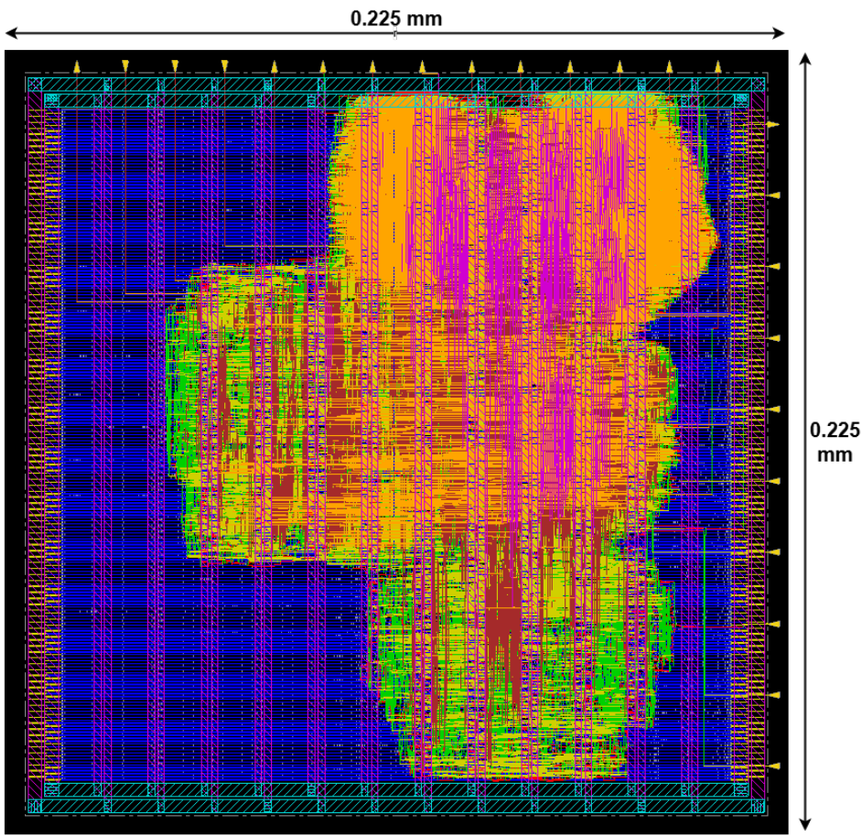

Figure 3.9: Optimized gate-level routing configuration ready for GDSII stream-out and *Signoff* verification.

The physical flow concludes with an intensive *Optimization-Routing* stage. This pass fine-tunes wire geometries to minimize dynamic power dissipation and maximize *Setup/Hold* timing margins. Following this final optimization step, the comprehensive manufacturing DRC suite is re-executed to guarantee that no geometrical violations were introduced during layout tuning.

The physical design is now fully integrated, and the final layout data can be exported via a standard GDSII stream file, which represents the industry-standard file format containing the exact geometric mask data required by the foundry for physical fabrication (Figure 3.9). Before fabrication tape-out, however, the GDSII netlist must undergo an *Signoff* verification phase. If *Signoff* Static Timing analysis, dynamic power reporting or *Power-Grid* (*IR-*

Drop) simulations detect any out-of-spec parameters, the digital flow must rollback to the appropriate implementation stage to resolve the issue.

3.4 Post-Layout *Signoff* and *Power-Analyses* Verification

This subsection delineates the comprehensive post-layout *Signoff* verification methodology deployed to validate the physical ASIC implementation of the Microprocessor architecture. To guarantee deterministic silicon behavior prior to tape-out, the layout is rigorously appraised across multiple performance vectors, categorized as follows:

- **Timing Analysis:** Validates path propagation delays against clock constraints.
- **Gate-Level Functional Verification (Post-Layout Behavioral Analysis):** Ensures logical-functional correctness under non-ideal physical delays.
- **Power Dissipation Estimation:** Quantifies static and dynamic power components.
- **Power Integrity and *IR-Drop* Analysis:** Maps the current density distribution across the *Power-Grid*.

All *Signoff* simulations are comprehensively executed across the entire MCMM matrix to secure robustness against environmental and process fluctuations.

Prior to structural electrical evaluation, a battery of physical manufacturability checks is enforced to guarantee geometric integrity. This includes deep Design Rule Checking (DRC) to verify that the integrated layouts satisfy lithographic spacing boundaries, localized metal density evaluations to ensure mechanical compatibility, and the detection of process-induced antenna effects and physical short circuits.

During *Signoff* validation, the EDA engines compile hierarchical verification reports. These data structures allow the designer to audit the design metrics with variable granularities, scaling from full chip-level validation down to individual standard-cell logic instances.

Furthermore, the *Signoff* infrastructure generates characterized macro models (e.g., .lib, .lef) that encapsulate the electrical, physical, and timing boundaries of the Microprocessor block across all target MCMM corners. These macro views enable seamless block-level integration with external peripherals, memory subsystems, and the primary padring layout. Crucially, these abstracted models reduce the computational complexity and memory footprint

during full-chip, top-level tape-out validation passes without compromising *Signoff* accuracy.

This multi-view characterization strategy is fundamental to mitigating scheduling risks during the final chip-assembly phases, where flat chip-level verifications frequently incur prohibitive computational run-times that compromise tight project deadlines.

Initial *Signoff* assessments isolate the clock distribution architecture to verify the physical execution of the *Clock Tree Synthesis*. Under the most critical operating corner defined by the slow-silicon, low-voltage, high-temperature boundary (SS, 0.81 V, 125°C) combined with worst-case parasitic interconnect RC models (Worst), the maximum clock *Skew* across the entire *Clock Tree* distribution converges to an absolute value of 18 ps.

Subsequently, full-chip static timing analysis is executed to ensure global constraint closure, explicitly measuring structural *Setup* and *Hold* timing margins against library-defined flip-flop windows. For *Setup-Time* analysis, the worst-case path exhibits a positive slack of 720 ps, occurring at the slow-silicon corner (SS, 0.81 V, 125°C, Worst). Conversely, the minimum *Hold-Time* margins converge to a positive slack of 86 ps, restricted by the fast-silicon, high-voltage, low-temperature corner (FF, 0.99 V, −40°C, Worst).

Upon achieving timing closure across all MCMM views, the finalized GDSII stream-out file and the corresponding structural gate-level Verilog netlist are compiled. This structural netlist is then ingested by subsequent *Signoff* verification engines to drive non-ideal functional and power-integrity passes.

Following timing *Signoff*, gate-level functional simulations are launched to validate the logical correctness of the mapped netlist. These behavioral verification runs target two primary operating points:

- **Typical Corner:** TT, 0.90 V, 25°C.
- **Fast Corner:** FF, 0.99 V, 125°C.

The *Typical* view validates nominal functional behavior under standard operating conditions. Conversely, the *Fast* corner evaluates the Microprocessor at maximum switching speeds, representing the worst-case boundary for behavioral correctness regarding *Hold* violations, thereby testing the architectural noise margins and structural resilience against race conditions.

Figure 3.10: Post-layout timing simulation waveforms exhibiting physical interconnect and cell delay non-idealities.

Post-layout behavioral analysis are executed utilizing the identical verification testbench framework deployed during pre- and post-*Synthesis* phases, ensuring that the executed firmware logic yields functionally equivalent output signatures. Compared to earlier logic-*Synthesis* simulations, these *Signoff*-level simulation runs incorporate high-fidelity physical attributes (Figure 3.10), as they ingest back-annotated Standard Delay Format (*.SDF*) data compiled from physical parasitic extractions and foundry-vetted Liberty (.lib) lookup tables.

During these gate-level simulation runs, the verification engine captures the real-time node *Switching-Activity*, exporting them into two distinct formats: Toggle Count Format (*.TCF*) and Value Change Dump (*.VCD*).

The *.TCF* file summarizes time-averaged *Switching-Activity* at a macro-block granularity, offering a compact file size optimized for rapid first-order power calculations. In contrast, the *.VCD* format logs the exact *Switching-Activity* and logical states of every standard-cell pin throughout the simulation execution. While the high fidelity of the *.VCD* file incurs significant computational overhead and storage footprints, its fine-grained resolution is mandatory for localized dynamic power calculations and high-accuracy power-integrity validation passes.

The accuracy of both the *.TCF* and *.VCD* activity models is bounded by the stimulus coverage provided by the simulation testbench. Consequently, the verification testbench must execute high-coverage assembly sequences to thoroughly exercise all internal execution pipelines, register files, and control logic blocks. Insufficient stimulus profiles risk masking high-current vectors, local clock anomalies, or extreme *Power-Grid* degradation loops, rendering post-layout behavioral and *Power Analyses* blind to localized failures. The underlying algorithmic structures of these functional verification suites will be detailed in Chapter 4.

Power dissipation and electrical distribution profiles are evaluated using two distinct EDA methodologies that feature varying degrees of accuracy and computational cost:

- **Vector-Less Power-Analysis:** Uses probabilistic algorithms driven by static, designer-defined *Switching-Activity* attributes.
- **Vector-Based Power Analysis:** Uses deterministic dynamic simulations driven by time-resolved input activity files (the *.VCD* stream).

This work utilizes both *Vector-Less* and *Vector-Based* paradigms to fully characterize the Microprocessor. Furthermore, to validate the localized voltage drops via static and dynamic *IR-Drop* simulations, the geometric connectivity between the internal physical *Power-Grid* and the external chip padring structure must be explicitly defined.

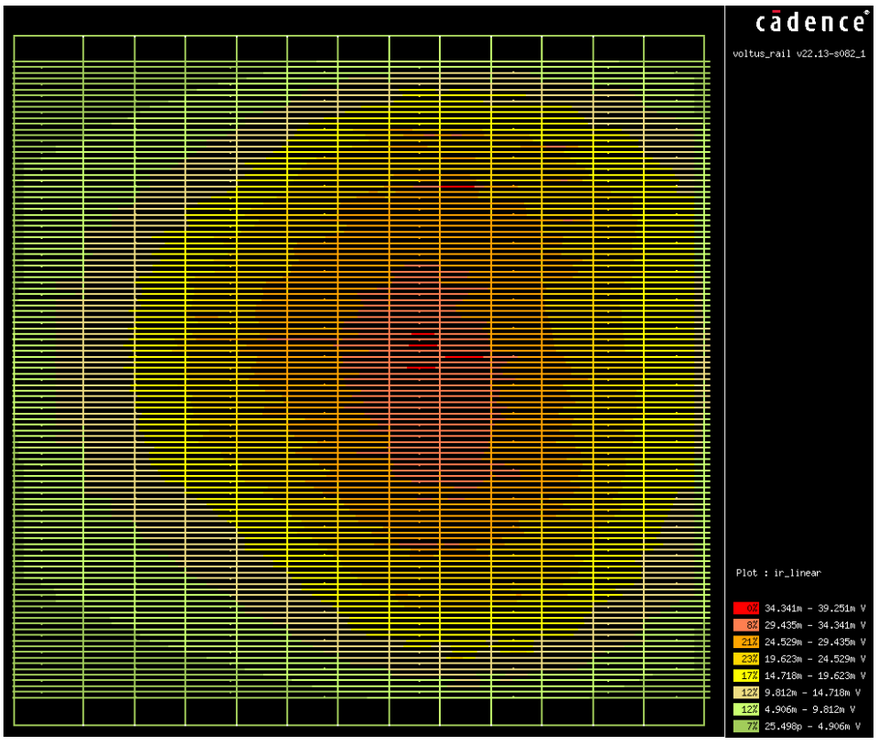

Figure 3.11: *Vector-Based* dynamic *IR-Drop* color-coded heatmap showing localized voltage drop distributions across the core *Power-Grid*.

Figure 3.11 illustrates the multi-rail current distribution derived from a deterministic *Vector-Based* dynamic power analysis. Leveraging the visual color-coded heatmap scale, designers can isolate high current-density coordinates to mitigate localized *IR-Drop* violations exceeding the maximum design limit constraint of 100 mV.

Vector-Less power passes were executed by combining the baseline structural statistics defined at the start of the *Place&Route* phase with the time-averaged activity metrics enclosed in the post-*Synthesis* $.TCF$ file. By setting the global *Toggle-Rate* to 0.5 and the *Increase Switch Consumption* overhead factor to 0.3, the *Vector-Less* analysis simulates a heavily over-constrained execution scenario. This pessimistic bounding box ensures that the *Power-Grid* maintains structural margins under radiation-induced single-event transients. Concurrently, the $.TCF$ parameters provide an initial realistic benchmark of nominal power dissipation profiles.

Conversely, deterministic *Vector-Based* simulations are leveraged to quantify precise spatial power dissipation and execute a comprehensive parametric characterization of the hardware. By mapping the full temporal detail of the $.VCD$ trace, this abstraction of verification complexity inside the testbench executed to verify the circuit.

Consolidating the verification logic within a unified testbench environment allows the exact same software-driven test sequences to be executed seamlessly across both pre-silicon EDA simulators and physical post-silicon laboratory test fixtures. This continuity enables high-resolution hardware characterization,

allowing for direct correlations between pre-tapeout power simulations and physical laboratory measurements.

Because a programmable Microprocessor is inherently a general-purpose processor, it is impossible to define a single worst-case operational application or a finite subset of instructions that guarantees absolute test coverage. Therefore, establishing a direct functional cross-correlation between simulated layout models and physical silicon data is crucial. Integrating an on-chip *Debug-Module* enables the step-by-step verification of every intermediate logical state (*Logical End-Point*) and architectural register checkpoint during the active execution of embedded firmware.

The combination of an integrated *Debug-Module* and deterministic *Vector-Based Signoff* simulations provides a robust pipeline for validating the Microprocessor across both virtual layout environments and automated laboratory instruments. Furthermore, this infrastructure enables the fine-grained characterization of the instruction-level energy profile, quantifying the average power dissipation for each individual instruction defined within the Reduced Instruction Set Computer - Five (RISC-V) Instruction Set Architecture (ISA).

For space-constrained Microprocessor implementations, this instruction-level power characterization is highly valuable, as it allows embedded developers to mathematically calculate the total energy consumption of compiled firmware binaries prior to deployment.

3.5 Microprocessor Physical Die Realization and Implementation Parameters

This subsection presents the physical ASIC die realization of the integrated Microprocessor utilizing the TSMC 28 nm CMOS Bulk HPC+ process node. Furthermore, it consolidates the primary metrics, structural constraints, and configuration parameters deployed across the entire digital implementation framework.

Upon verifying physical manufacturability, architectural equivalence, and constraint closure across all MCMM views, the structural core layout was ready for final physical integration. In this specific implementation vehicle, the Microprocessor was co-integrated onto a shared multi-project silicon *Die* alongside independent subsystem blocks.

Following the hierarchical assembly of all sub-blocks into a single unified top-level layout file, a final validation pass was initiated. This top-level *Signoff* phase re-executed exhaustive manufacturing DRC and gate-level functional simulations to validate the structural integrity of the Microprocessor core when physically and electrically bound to the primary *I/O Padring* network.

Upon successful clearance of these top-level integration checks, the finalized

database was streamed out and delivered to *Europractice*. Acting as the structural multi-project intermediation service interface, *Europractice* managed the physical manufacturing and layout tape-out pipeline with the TSMC foundry to realize the physical silicon Microprocessor.

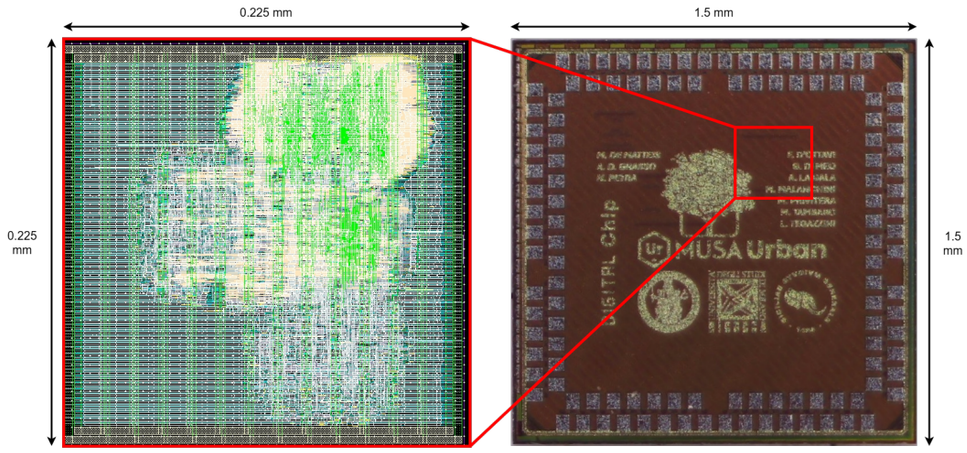

Figure 3.12: Integrated Microprocessor final layout design (left) and corresponding microphotograph of the physical silicon die (right).

Figure 3.12 illustrates the physical outcomes of the tape-out process. The left view depicts the finalized physical layout submitted to the foundry for mask generation. Conversely, the right view presents the optical microscope photograph of the fabricated silicon *Die*, displaying the physical metallization layer with the Multilayered Urban Sustainability Action (MUSA) logo and the inscriptions honoring the hardware development team responsible for this multi-project integration.

Table 3.3: Gate Equivalent of Main Components of Microprocessor.

Main Components	**Gate Equivalent [GE]**
I/O-Module	1,307
Debug-Module	2,051.5
Code-Memory	11,138.5
Data-Memory	16,356.5
RV32I-Core	13,649
Routing and Peripheral Circuit	8,179.75
Total	52,682.25

To evaluate the structural density of the architecture independently of the specific TSMC 28 nm process dimensions, the physical complexity of the layout is quantified using the Gate Equivalents (GE) metric. The *Gate Equivalent* abstractly standardizes area metrics across disparate technology nodes, enabling architectural density comparisons with structurally similar Microprocessors fabricated on alternative process nodes. By industry convention, a single *Gate Equivalent* (1 GE) is defined as the physical area occupied by a minimum-sized inverter cell,within the target technology library.

Table 3.3 details the exact *Gate Equivalent* allocation parsed across the primary macro-blocks of the Microprocessor. This matrix isolates the active cell area consumed by core execution logic from the passive routing area overhead required to route complex signal buses and control networks between the macro-blocks, culminating in the total *Gate Equivalent* count demanded by the integrated hardware footprint.

Table 3.4: Microprocessor Implementation Parameters.

Parameters	**Value**	**Units**
Clock Frequency	100	**[MHz]**
Technology	28nm CMOS Bulk HPC+ TSMC	**[-]**
Microprocessor Size	0.225 × 0.225	**[mm]**
RV32I-Core Size	0.097 × 0.104	**[mm]**
Data-Memory Capacity	128	**[Byte]**
Code-Memory Capacity	128	**[Byte]**
Static-to-Dynamic Ratio	0.2	**[-]**
Vertical Power Stripes Pitch	14.165	**[μm]**
Power Stripe Width	2 (M7)	**[μm]**
Power Ring Width	4 (M7, M8)	**[μm]**
Temperature	-40, 25, 125	**[°C]**
Process	SS, SF, FS, FF, TT	**[-]**
Digital Cell Threshold Voltage	HVT, LVT, SVT	**[-]**
Supply Voltage (V_{DD})	0.81, 0.9, 0.99	**[V]**
Setup-Time (Worst, SS 0.81V 125°C)	720	**[ps]**
Hold-Time (Worst, FF 0.99V -40°C)	86	**[ps]**
Clock Skew (Worst, SS 0.81V 125°C)	18	**[ps]**
Clock Skew (Worst, SS 0.81V 125°C) to Clock Period	0.18	**[%]**

Table 3.4 summarizes the execution parameters, active MCMM analysis views, and physical design choices applied during the physical *Synthesis* and *Place&Route* optimization routines, alongside the definitive worst-case timing slack margins extracted during the final *Signoff* phase.

Chapter 4

Verification and Characterization Methodology

This chapter details the verification methodologies, architectural compliance frameworks, and experimental characterization procedures deployed to validate the Microprocessor from its initial behavioral abstraction down to its finalized post-layout Application-Specific Integrated Circuits (ASIC) implementation. Furthermore, this section highlights the specific testing frameworks utilized to guarantee strict architectural compliance with the official Reduced Instruction Set Computer - Five (RISC-V) Instruction Set Architecture (ISA) specification.

A Microprocessor is, by definition, a reprogrammable general-purpose circuit. Due to this intrinsic programmable nature, there is no predefined average or worst-case application scenario, which makes the verification phase particularly challenging.

For general-purpose Microprocessors and reconfigurable fabrics designed to accommodate a diverse spectrum of target applications, the verification and characterization phase becomes exceptionally complex. It demands a rigorous co-design of both the pre-silicon simulation environments and the post-silicon laboratory test fixtures.

Moreover, this broad operational variability requires a strategic analysis of the validation algorithms to ensure maximum structural and toggling coverage across the internal logic, stimulating the highest number of architectural use-cases with the minimum number of discrete test vectors.

The most deterministic method to guarantee full coverage of the logical-functional behavior of the circuit with absolute certainty is to exhaustively test every possible combination of instructions and memory states. The total number of unique, architecturally valid instruction sequences, denoted as $\mathcal{F}_{\text{total}}$, capable of being hosted within an N-word instruction memory defines the functional state space cardinality and can be expressed by the following combinatorial formula (Eq. 4.1):

$$\mathcal{F}_{total} = \left(n_R 2^{15} + (n_I + n_S + n_B)2^{22} + (n_U + n_J)2^{25}\right)^N \tag{4.1}$$

This metric quantifies the computational state space of the proposed RV32I compliant Microprocessor core. The parameters within the equation (Eq. 4.1) reflect the specific subset of the RV32I ISA implemented in this design, excluding the *FENCE* instruction. The formulation incorporates the exact number of distinct opcodes and function codes (*funct3*/*funct7*) mapped per instruction format, alongside the complete combinatorial space defined by the 5-bit register fields (*rs1*, *rs2*, *rd*) and their respective immediate fields:

- $\boldsymbol{n\_R = 10}$**:** Register-to-register arithmetic/logic instructions.
- $\boldsymbol{n\_I = 18}$**:** Immediate-type instructions, including Loads and $JALR$.
- $\boldsymbol{n\_S = 3}$**:** Store instructions (SB, SH, SW).
- $\boldsymbol{n\_B = 6}$**:** Conditional branch instructions.
- $\boldsymbol{n\_U = 2}$**:** Upper-immediate instructions (LUI, $AUIPC$).
- $\boldsymbol{n\_J = 1}$**:** Unconditional jump instruction (JAL).

Given these architectural constraints, the single-instruction variability yields approximately 2.142×10^8 valid hardware encodings. For the constrained instruction memory depth of $N = 32$ words configured in this specific implementation vehicle, the total functional state space expands to (Eq. 4.2):

$$\mathcal{F}_{total} = \left(10 \cdot 2^{15} + 27 \cdot 2^{22} + 3 \cdot 2^{25}\right)^{32} \approx 3.74 \times 10^{266} \tag{4.2}$$

This massive value defines the algorithmic entropy of the system, representing the specific subset of the 2^{1024} physical bit-combinations that constitute valid executable machine code for the Microprocessor. Even for the reduced memory footprint of this test vehicle, the resulting complexity yields $\mathcal{F}_{\text{total}} \approx 3.74 \times 10^{266}$. Consequently, a brute-force verification approach is computationally infeasible, particularly as the physical memory capacity scales.

For this reason, it is mandatory to implement a structured matrix of advanced testbench procedures and high-coverage verification firmware. This dual framework enables deep layout and logical validation without the necessity of exhaustively scanning the mathematically intractable functional state space.

The verification hierarchy utilized across the three primary milestones of the design progressing from initial behavioral description to physical ASIC tape-out is schematically summarized in Figure 4.1.

Verification vectors are initially executed within a purely logical simulation environment (Siemens Electronic Design Automation (EDA) ModelSim) to debug and validate the Microprocessor core and Register-Transfer Level (RTL) functionality. Upon achieving functional correctness, the design is mapped onto a hardware-emulation platform using an Field Programmable Gate Arrays (FPGA) prototyping flow (AMD Xilinx Vivado). This phase re-executes the

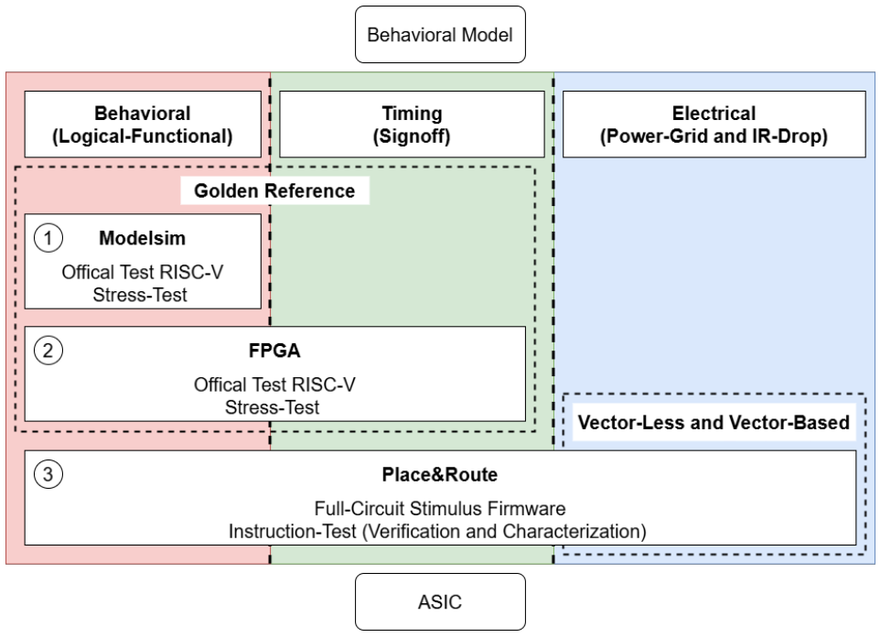

Figure 4.1: Block diagram of the hierarchical multi-stage verification framework spanning RTL behavioral simulation, FPGA emulation, and post-layout gate-level ASIC *Signoff*.

complete test suite on physical hardware, providing a high-speed emulation environment. Furthermore, because the FPGA implementation closer mirrors real physical structures, it serves as a preliminary platform for hardware timing evaluation and initial board-level verification.

The final phase delineated in Figure 4.1 is executed at the conclusion of the digital physical design flow (Cadence Genus and Innovus). This post-*Place&Route* gate-level verification layer provides high-fidelity behavioral, static timing, and electrical *Signoff* analyses.

Through this multi-tiered, progressive verification infrastructure, the Microprocessor architecture is thoroughly validated, characterized and hardened prior to physical silicon manufacturing.

4.1 Behavioral Functional-Logic and First-Order Timing Verification

This subsection details the verification routines and design-phase frameworks established to validate the logical-functional execution and to extract first-order timing performance metrics of the Microprocessor core.

As illustrated in the hierarchical pipeline (Figure 4.1), the initial pre-silicon verification loop relies on two sequential platforms: RTL simulation via Siemens EDA ModelSim and physical emulation via Field Programmable Gate Arrays (FPGAs). ModelSim serves as the primary Integrated Development Environment (IDE) for structural register-transfer level (RTL) definition and logic design. At this abstraction layer, simulation passes focus exclusively

on evaluating logical-functional behavior under idealized timing conditions, detached from physical hardware non-idealities.

Conversely, the FPGA prototyping platform introduces non-ideal hardware constraints, eliminating the abstraction of the initial software-only simulation phase. Implementing the architecture on a physical programmable fabric allows the designer to execute a first-order static timing analysis, isolating layout-dependent critical paths and *Setup/Hold* hazards early in the design cycle.

Crucially, both the RTL simulation and FPGA emulation passes serve as diagnostic design-aids, providing rapid feedback loops to detect structural design bugs, race conditions or poorly synthesized hardware descriptions (i.e., the translation of the behavioral Hardware Description Language (HDL) code into elementary logic gates or Look-Up Table (LUT)s).

The validation algorithms selected to drive these two initial verification environments comprise the official RISC-V architectural test [85] suite alongside customized, high-coverage *Stress-Tests*.

The official RISC-V architectural tests provide an initial validation of the core execution unit and verify that the implementation complies strictly with the international RISC-V ISA specification. These test compliance suites systematically exercise every discrete instruction defined within the implemented target ISA.

Environmental limits, however, dictate that the official RISC-V architectural test suite requires a memory footprint that exceeds the physical allocations configured for this specific ASIC test vehicle. Consequently, during the RTL simulation and FPGA emulation phases, the storage arrays (*Data-Memory* and *Code-Memory*) were temporarily expanded to accommodate the full compliance binary images.

Crucially, standard compliance test suites are inherently insufficient to guarantee absolute core correctness. This limitation arises because the executed validation firmware relies on the same internal instruction decoder and execution unit under test to report its own completion status. Consequently, if multiple complementary instruction classes are erroneously implemented, their faults can mask one another, leaving deep hardware bugs undetected by the official compliance framework.

To mitigate this visibility risk, a dedicated array of functional *Stress-Tests* was engineered. These targeted software blocks validate the execution of each instruction by intentionally targeting worst-case edge cases, maximum and minimum operand boundaries, and corner-case sign-extension or overflow transitions across the full dynamic range of the data path.

Both the official compliance suites and the custom *Stress-Tests* are evaluated using a *Golden-Reference* verification paradigm [86–88]. This technique compares the execution traces of the Device Under Test (DUT) against a

pre-validated, golden-standard software model.

An architectural model compliant with the official RISC-V specification was deployed as the *Golden-Reference*. Because the RISC-V ISA defines an instruction-set abstraction rather than a rigid hardware execution pipeline, a literal bit-for-bit hardware trace comparison with the golden model is impossible. To resolve this structural asymmetry, a memory-remapping routine was engineered to synchronize and intercept the state transitions of both platforms at every discrete clock cycle during firmware execution.

The internal register files and memory arrays constitute the primary logical endpoints that define the state space of the Microprocessor. Relying strictly on the final execution code or a singular pass/fail output pin can easily mask internal transient errors or data corruption loops within intermediate registers. By evaluating the state space of these logical endpoints concurrently with firmware execution, the verification environment ensures absolute behavioral equivalence, maximizing functional coverage across all primary use cases while confirming compliance with the RISC-V standard.

4.2 Post-Layout Behavioral Verification, *Signoff* and *Power-Analysis*

This subsection elucidates the structured verification and hardware characterization pipeline executed within the digital implementation flow utilizing Cadence Genus and Innovus.

By mapping the high-level HDL onto standard foundry logic cells, the digital implementation flow enables high-fidelity physical simulations that closely match actual post-silicon behavior. Leveraging foundry-vetted standard-cell library views and continuous parasitic extraction metrics, the design is audited under diverse process, voltage, and temperature configurations (Multi-Corner Multi-Mode (MCMM) operating corners).

Consequently, the verification passes executed at this level specifically during the final static timing *Signoff* and power-integrity analysis phases are critical. They remove the logical abstractions inherent to the preceding pre-silicon verification environments, providing a deterministic model of physical hardware behavior.

Because these layout-accurate evaluations resolve cell-level geometric realities, they demand significantly greater computational bandwidth and extended EDA engineering execution run-times.

The post-layout validation suite concurrently evaluates the design across three main domains: gate-level logical-functional behavior (post-layout *Behavioral Analysis*), deterministic constraint closure (timing *Signoff*), and physical electrical integrity (*Power-Grid* and *IR-Drop* profiling).

Post-layout behavioral simulations are driven by a specialized firmware payload engineered to maximize internal toggle coverage, supplemented by probabilistic activity configurations (global *Toggle-Rate* and *Increase Switch Consumption* factors) established at the start of the *Place&Route* phase. This validation step ensures that the netlist preserves strict logical-functional equivalence throughout the structural transformations of the digital flow. The equivalence checks are systematically executed across three key milestones: pre-*Synthesis* RTL, post-*Synthesis* netlist, and post-*Place&Route* finalized layout.

To satisfy the stringent area constraints of the physical silicon die allocation, the storage footprints of the *Data-Memory* and *Code-Memory* arrays were reduced, which prevents the direct execution of the official RISC-V compliance test images. However, because the RISC-V ISA specification does not dictate a minimum legal physical memory capacity, scaling down these memory depth parameters does not alter the underlying execution logic or register topology. The core implementation therefore maintains absolute architectural compliance with the RISC-V standard.

To ensure strict validation, the verification firmware must achieve a high toggle rate across all internal functional blocks and control buses. Given the 40 distinct instructions that compose the base RV32I ISA, executing a flat, unrolled sequence of all operations within the scaled memory constraints is unfeasible, requiring optimization.

Leveraging loop structures, unconditional jumps, and conditional branches, the verification software implements dense, cyclic execution loops that maximize hardware activity within a compact instruction footprint. Furthermore, since distinct instruction classes share underlying hardware datapath execution resources, a highly optimized subset of instructions can be isolated for test encapsulation. For instance, validating a *Load Word* (LW) instruction intrinsically exercises the identical decoding paths, sign-extension networks, and register-file write ports utilized by lesser memory operations such as *Load Byte* (LB) and *Load Half Word* (LH).

Firmware Table 4.1 illustrates a segment of the targeted assembly sequence compiled to drive the post-*Place&Route* simulation testbench. Following the completion of the post-*Place&Route* simulation pass, the EDA environment logs the exact nodal switching frequencies, exporting the activity into Toggle Count Format (*.TCF*) and Value Change Dump (*.VCD*) database streams.

By ingesting these activity files into the *Signoff* engines, the framework precisely models cell-level timing margins and *Power-Grid* degradation loops, enabling a rigorous, layout-accurate characterization of the Microprocessor.

The timing *Signoff* routines evaluate the physical layout to ensure rigid constraint closure across all operating corners defined within the MCMM matrix. This analysis evaluates structural *Setup-Time* margins, *Hold-Time*

Table 4.1: Assembly testbench code segment for post-layout functional and power-integrity validation.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00C00393	addi x7, x0, 12	Load Imm 12 into register x7.
00638413	addi x8, x7, 6	Add Imm 6 to x7; store result in x8.
407404B3	sub x9, x8, x7	Subtract x7 from x8; store result in x9.
0093A623	sw x9, 12(x7)	Store 32-bit word from x9 into memory at x7 + 12.
00748863	beq x9, x7, 16	Branch to PC + 16 if register x9 equals x7.
00348493	addi x9, x9, 3	Add Imm 3 to x9; store result in x9.
00150513	addi x10, x10, 1	Increment iteration counter x10 by 1.
FF5FF0EF	jal x1, -12	Unconditional jump to PC - 12; save return in x1.
10450513	addi x10, x10, 260	Add Imm 260 to x10 for peripheral offset tracking.
00A38823	sw x10, 16(x7)	Store 32-bit word from x10 into memory at x7 + 16.
00741533	sll x18, x8, x7	Shift left logical x8 by amount in x7; store in x18.
00339613	slli x12, x7, 3	Shift left logical x7 by Imm 3; store in x12.
05448493	addi x9, x9, 84	Add Imm 84 to x9 for boundary memory alignment.
00C4C6B3	xor x13, x9, x5	Bitwise XOR of x9 and x5; store result in x13.
02069463	bne x13, x6, 40	Branch to PC + 40 if register x13 is not equal to x6.
00642583	lw x11, 6(x8)	Load 32-bit word from memory at x8 + 6 into x11.
009583B3	add x7, x11, x9	Add x11 to x9; store runtime hardware result in x7.
00401283	lh x5, 4(x0)	Load signed 16-bit halfword from memory 4 into x5.
00501523	sh x5, 10(x0)	Store 16-bit halfword from x5 into memory 10.
00100303	sb x0, 1(x0)	Store 8-bit zero value from x0 into target memory 1.
006006A3	lw x13, 6(x0)	Load 32-bit word from memory 6 into x13.
00802803	lw x16, 8(x0)	Load 32-bit word from memory 8 into x16.
00C02883	lw x17, 12(x0)	Load 32-bit word from memory 12 into x17.
01180A63	beq x16, x17, 20	Branch to PC + 20 if register x16 equals x17.
09700513	addi x10, x10, 151	Add Imm 151 to x10 for routine data-path scaling.
01000293	addi x15, x0, 16	Load Imm 16 into register x15.
00A2A023	sw x15, 0(x5)	Store 32-bit word from x15 into memory at x5.
00008067	jalr x0, 0(x1)	Return from function call by branching to in x1.
06100513	addi x10, x10, 97	Add Imm 97 to counter x10 for base tracking.
01000293	addi x15, x0, 16	Load constant loop bound 16 into register x15.
00A2A023	sw x15, 0(x5)	Store 32-bit word from x15 into memory at x5.

requirements, and the localized clock *Skew* across the *Clock Tree Synthesized* network.

Concurrently, the electrical integrity passes map global current density profiles and isolate maximum *IR-Drop* violations across the physical *Power-Grid*. These evaluations leverage both *Vector-Less* (probabilistic/static) and *Vector-Based* (dynamic, using the simulated $.TCF$ and $.VCD$ trace data) power analysis methodologies. Both techniques independently extract the cumulative power dissipation vectors and map the voltage degradation profile across the internal *Power-Grid*.

4.3 Firmware for Instruction-Level Power Characterization

This subsection delineates the programmatic algorithms and custom firmware architectures engineered to execute high-resolution power characterization of the general-purpose Microprocessor core.

To properly model and characterize a flexible, programmable Microprocessor, a rigorous instruction-level power analysis framework must be established. The

static and dynamic energy consumption metrics of the hardware are mapped by quantifying the precise current dissipation vector associated with every discrete operation in the RISC-V ISA.

Leveraging dynamic, *Vector-Based* EDA simulations driven by back annotated *.TCF* and *.VCD Switching-Activity* databases, the physical complexity of the layout verification is shifted directly into the software compilation layer of the verification testbench. When combined with the dedicated on-chip *Debug-Module*, this unified validation architecture enables identical execution sequences to be applied across both virtual EDA simulators (Cadence Innovus) and physical post-silicon laboratory test fixtures. This methodological continuity ensures a highly correlated hardware characterization loop between pre-tapeout models and physical silicon measurements.

To extract statistically significant power metrics, each discrete instruction execution must be replicated across an extended, steady-state temporal window. For this purpose, an array of 37 distinct assembly firmware routines was synthesized, with each program dedicated to a specific arithmetic, logic, memory, or control flow operation.

These assembly routines are structured as infinite execution loops. By software convention, the 32nd instruction slot corresponding to the maximum upper boundary of the allocated instruction memory hosts an unconditional jump instruction (*JAL*) targeting the primary execution index of the loop.

This infinite loop structure simplifies post-silicon laboratory measurements by eliminating complex external triggering mechanisms or tight timing synchronization windows for test isolation. Within the pre-tapeout EDA environment, *Power Analysis* passes evaluate a continuous window of 10,000 clock cycles, executing an equivalent volume of 10,000 instructions under active switching conditions.

The compilation of each characterization program is driven by three main optimization vectors:

- The structural logic function implemented by the specific execution unit.
- The data operand variability injected into the pipeline.
- The *Hamming-Distance* separating successive operations.

The *Hamming-Distance* defines the exact number of concurrent bit-flips transitioning between sequential operational states. Together, the operand entropy and the instruction-to-instruction *Hamming-Distance* determine the total localized *Switching-Activity* generated within the core datapath. These parameters are balanced to induce a nominal, time-averaged switching profile. By avoiding optimal or worst-case switching extremes, the characterization yields an accurate baseline of nominal current dissipation rather than an unrepresentative performance bound.

A dedicated standalone routine (*Loop-Firmware*) was compiled to isolate the baseline power overhead introduced exclusively by the JAL instruction during *Loop-Back* transitions. By isolating this operational coefficient, the true average current consumption of any single target instruction can be extracted via a weighted algebraic subtraction.

Conversely, standard synchronization and exception instructions such as $FENCE$, $ECALL$, and $EBREAK$ which place the execution pipeline into an idle or halted state, are characterized using a dedicated quiescent routine (*Blank-Firmware*). This program runs a zero-activity sequence to map the absolute minimum sleep-state current floor of the Microprocessor.

The following sections illustrate representative assembly examples for each primary instruction format defined within the RV32I ISA. The structural concepts applied to these single instances are identically mapped across all remaining operations within their respective instruction classes. (Note: The complete codebases for unlisted instructions, including the baseline *Loop-Firmware* and *Blank-Firmware* routines, are compiled in Appendix B).

Table 4.2: Firmware characterization loop for the $ADDI$ (*I-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
03908093	addi x1, x1, 57	Add Imm 57 (0x39) to register x1.
00210113	addi x2, x2, 2	Add Imm value 2 to register x2.
00318193	addi x3, x3, 3	Add Imm value 3 to register x3.
0B220213	addi x4, x4, 178	Add Imm value 178 to register x4.
00528293	addi x5, x5, 5	Add Imm value 5 to register x5.
00630313	addi x6, x6, 6	Add Imm value 6 to register x6.
26E18193	addi x3, x3, 622	Add Imm value 622 to register x3.
00840413	addi x8, x8, 8	Add Imm value 8 to register x8.
00948493	addi x9, x9, 9	Add Imm value 9 to register x9.
00A50513	addi x10, x10, 10	Add Imm value 10 to register x10.
00B58593	addi x11, x11, 11	Add Imm value 11 to register x11.
51A28293	addi x5, x5, 1306	Add Imm value 1306 to register x5.
00D68693	addi x13, x13, 13	Add Imm value 13 to register x13.
00E70713	addi x14, x14, 14	Add Imm value 14 to register x14.
51A28293	addi x5, x5, 1306	Add Imm value 1306 to register x5.
01080813	addi x16, x16, 16	Add Imm value 16 to register x16.
73868693	addi x13, x13, 1848	Add Imm value 1848 to register x13.
01290913	addi x17, x17, 18	Add Imm value 18 to register x17.
13298993	addi x9, x19, 306	Add Imm value 306 to x19; store result in x9.
1B4A0A13	addi x20, x20, 436	Add Imm value 436 to register x20.
2AE28293	addi x5, x30, 686	Add Imm value 686 to x30; store result in x5.
016B0B13	addi x22, x22, 22	Add Imm value 22 to register x22.
1B7B8B13	addi x23, x23, 439	Add Imm value 439 to register x23.
018C0C13	addi x24, x24, 24	Add Imm value 24 to register x24.
019C8C13	addi x25, x25, 25	Add Imm value 25 to register x25.
01AD0D13	addi x26, x26, 26	Add Imm value 26 to register x26.
01BD8D13	addi x27, x27, 27	Add Imm value 27 to register x27.
2AE28293	addi x5, x30, 686	Add Imm value 686 to x30; store result in x5.
1BD68693	addi x13, x13, 445	Add Imm value 445 to register x13.
1BE70713	addi x14, x14, 446	Add Imm value 446 to register x14.
01F58593	addi x11, x11, 31	Add Imm value 31 to register x11.
F85FF06F	jal x0, -120	Unconditional relative jump to PC - 120.

Listing Table 4.2 presents the assembly sequence designed to isolate the average current consumption of the register-immediate $ADDI$ operation. The initial 31 instructions loaded sequentially into the instruction memory execute

arithmetic additions against varying immediate fields, structured to balance datapath entropy.

Operations are interleaved between identical target destinations and alternating registers. As the infinite loop repeats, the underlying values within the standard *Register-File* shift continuously, maintaining a stable, representative switching profile across the execution loop. This same algorithmic structure governs the characterization of upper-immediate (*U-Type*) instructions like LUI and $AUIPC$.

Table 4.3: Firmware characterization loop for the ADD (*R-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
37B08093	addi x1, x1, 891	Add Imm 891 (0x37B) to register x1.
7FF10113	addi x2, x2, 2047	Add Imm 2047 (0x7FF) to register x2.
70F10113	addi x2, x2, 1807	Add Imm 1807 (0x70F) to register x2.
002080B3	add x1, x1, x2	Add register x2 to x1; store result in register x1.
01F080B3	add x1, x1, x31	Add register x31 to x1; store result in register x1.
001080B3	add x1, x1, x1	Add register x1 to x1; store result in register x1.
01F08FB3	add x31, x1, x31	Add register x31 to x1; store result in register x31.
00208FB3	add x31, x1, x2	Add register x2 to x1; store result in register x31.
01F18FB3	add x31, x3, x31	Add register x31 to x3; store result in register x31.
00110133	add x4, x18, x1	Add register x1 to x18; store result in register x4.
01F10133	add x4, x18, x31	Add register x31 to x18; store result in register x4.
01F080B3	add x1, x1, x31	Add register x31 to x1; store result in register x1.
002080B3	add x1, x1, x2	Add register x2 to x1; store result in register x1.
00110133	add x4, x18, x1	Add register x1 to x18; store result in register x4.
01110133	add x4, x18, x17	Add register x17 to x18; store result in register x4.
01F08FB3	add x31, x1, x31	Add register x31 to x1; store result in register x31.
00208FB3	add x31, x1, x2	Add register x2 to x1; store result in register x31.
01F18FB3	add x31, x3, x31	Add register x31 to x3; store result in register x31.
00110133	add x4, x18, x1	Add register x1 to x18; store result in register x4.
01F10133	add x4, x18, x31	Add register x31 to x18; store result in register x4.
01F080B3	add x1, x1, x31	Add register x31 to x1; store result in register x1.
002080B3	add x1, x1, x2	Add register x2 to x1; store result in register x1.
01F08FB3	add x31, x1, x31	Add register x31 to x1; store result in register x31.
00208FB3	add x31, x1, x2	Add register x2 to x1; store result in register x31.
01F18FB3	add x31, x3, x31	Add register x31 to x3; store result in register x31.
00110133	add x4, x18, x1	Add register x1 to x18; store result in register x4.
01F10133	add x4, x18, x31	Add register x31 to x18; store result in register x4.
01F080B3	add x1, x1, x31	Add register x31 to x1; store result in register x1.
002080B3	add x1, x1, x2	Add register x2 to x1; store result in register x1.
01F08FB3	add x31, x1, x31	Add register x31 to x1; store result in register x31.
00208FB3	add x31, x1, x2	Add register x2 to x1; store result in register x31.
F91FF06F	jal x0, -110	Unconditional relative jump to PC - 110.

Listing Table 4.3 outlines the firmware engineered to evaluate the register-register ADD instruction. The initial three instructions execute immediate loads ($ADDI$) to seed specific bit patterns into the internal register file. These seed values are then processed by the subsequent 28 sequential ADD operations inside the infinite loop.

The arithmetic interactions are mapped to maintain a stable switching profile, continuously rotating the source and destination register addresses across the *Register File*. The instruction sequence covers diverse structural register topologies, ranging from orthogonal isolated register pairs to self-accumulating destinations. For this instruction class, the 32nd *Loop-Back*

instruction branches back to the first active *ADD* instruction, bypassing the initial register seeding block.

Table 4.4: Firmware characterization loop for the *SW* (*S-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00000297	auipc x5, 0	Add 0 to PC shifted left by 12 bits; store in x5.
03908093	addi x1, x1, 57	Add Imm 57 (0x39) to register x1.
0012A023	sw x1, 0(x5)	Store 32-bit word from x1 to memory M[x5 + 0].
0012A223	sw x1, 4(x5)	Store 32-bit word from x1 to memory M[x5 + 4].
0012A423	sw x1, 8(x5)	Store 32-bit word from x1 to memory M[x5 + 8].
0012A623	sw x1, 12(x5)	Store 32-bit word from x1 to memory M[x5 + 12].
0012A823	sw x1, 16(x5)	Store 32-bit word from x1 to memory M[x5 + 16].
0012AA23	sw x1, 20(x5)	Store 32-bit word from x1 to memory M[x5 + 20].
0012AC23	sw x1, 24(x5)	Store 32-bit word from x1 to memory M[x5 + 24].
0012AE23	sw x1, 28(x5)	Store 32-bit word from x1 to memory M[x5 + 28].
0212A023	sw x1, 32(x5)	Store 32-bit word from x1 to memory M[x5 + 32].
0212A223	sw x1, 36(x5)	Store 32-bit word from x1 to memory M[x5 + 36].
0212A423	sw x1, 40(x5)	Store 32-bit word from x1 to memory M[x5 + 40].
0212A623	sw x1, 44(x5)	Store 32-bit word from x1 to memory M[x5 + 44].
0212A823	sw x1, 48(x5)	Store 32-bit word from x1 to memory M[x5 + 48].
0212AA23	sw x1, 52(x5)	Store 32-bit word from x1 to memory M[x5 + 52].
0212AC23	sw x1, 56(x5)	Store 32-bit word from x1 to memory M[x5 + 56].
0212AE23	sw x1, 60(x5)	Store 32-bit word from x1 to memory M[x5 + 60].
0412A023	sw x1, 64(x5)	Store 32-bit word from x1 to memory M[x5 + 64].
0412A223	sw x1, 68(x5)	Store 32-bit word from x1 to memory M[x5 + 68].
0412A423	sw x1, 72(x5)	Store 32-bit word from x1 to memory M[x5 + 72].
0412A623	sw x1, 76(x5)	Store 32-bit word from x1 to memory M[x5 + 76].
0412A823	sw x1, 80(x5)	Store 32-bit word from x1 to memory M[x5 + 80].
0412AA23	sw x1, 84(x5)	Store 32-bit word from x1 to memory M[x5 + 84].
0412AC23	sw x1, 88(x5)	Store 32-bit word from x1 to memory M[x5 + 88].
0412AE23	sw x1, 92(x5)	Store 32-bit word from x1 to memory M[x5 + 92].
0612A023	sw x1, 96(x5)	Store 32-bit word from x1 to memory M[x5 + 96].
0612A223	sw x1, 100(x5)	Store 32-bit word from x1 to memory M[x5 + 100].
0612A423	sw x1, 104(x5)	Store 32-bit word from x1 to memory M[x5 + 104].
0612A623	sw x1, 108(x5)	Store 32-bit word from x1 to memory M[x5 + 108].
DA708093	addi x1, x1, -601	Add Imm -601 (0xDA7) to register x1.
F8DFF06F	jal x0, -116	Unconditional relative jump to PC - 116.

Listing Table 4.4 illustrates the firmware configuration utilized to characterize the *SW* (*Store Word*) instruction. The initial instructions load the target data payload and baseline memory address pointers into the register file. The subsequent store operations execute consecutive write cycles into the *Data-Memory*.

To prevent the data bus from settling into a static, zero-power state during infinite iterations, an *ADDI* instruction modifies the data register payload immediately prior to the *Loop-Back* branch. This architectural tweak ensures that the data written to memory varies dynamically with every loop iteration, maintaining active, non-zero toggling across the memory interface.

Although classified under the *I-Type* instruction encoding format, memory *Load* operations require a dedicated firmware (Table 4.5) that closely mirrors the dual-bus interactions of *Store* configurations.

The preamble instructions format a known data pattern inside the register file and write it directly into a specific memory address within the *Data-Memory* block. This setup allows the subsequent *LW* (*Load Word*) instructions to read valid data transitions from the memory.

Table 4.5: Firmware characterization loop for the *LW* (*I-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00000297	auipc x5, 0	Add 0 to PC shifted left by 12 bits; store in x5.
00028293	addi x5, x5, 0	Copy contents of register x5 into register x5 .
CAFEB537	lui x10, 0xCAFEB	Load 20-bit Imm 0xCAFEB into upper bits of x10.
DEADB5B7	lui x11, 0xDEADB	Load 20-bit Imm 0xDEADB into upper bits of x11.
00A2A023	sw x10, 0(x5)	Store 32-bit word from x10 to memory M[x5 + 0].
00B2A223	sw x11, 4(x5)	Store 32-bit word from x11 to memory M[x5 + 4].
0002A503	lw x10, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x10.
0042A583	lw x11, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x11.
0002A603	lw x12, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x12.
0042A683	lw x13, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x13.
0002A703	lw x14, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x14.
0042A783	lw x15, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x15.
0002A803	lw x16, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x16.
0042A883	lw x17, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x17.
0002A903	lw x18, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x18.
0042A983	lw x19, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x19.
0002AA03	lw x20, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x20.
0042AA83	lw x21, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x21.
0002AB03	lw x22, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x22.
0042AB83	lw x23, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x23.
0002AC03	lw x24, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x24.
0042AC83	lw x25, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x25.
0002AD03	lw x26, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x26.
0042AD83	lw x27, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x27.
0002AE03	lw x28, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x28.
0042AE83	lw x29, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x29.
0002AF03	lw x30, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x30.
0042AF83	lw x31, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x31.
0042AE83	lw x29, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x29.
03908093	addi x1, x1, 57	Add Imm 57 (0x39) to register x1.
0012A023	sw x1, 0(x5)	Store 32-bit word from x1 to memory M[x5 + 0].
F9DFF06F	jal x0, -100	Unconditional relative jump to PC - 100.

Similar to the *Store* characterization methodology, the underlying memory data payload is iteratively modified and re-written to the memory location before loop execution. This mechanism ensures that the read data bus toggles continuously with every iteration, preventing static data stabilization.

Listing Table 4.6 presents the verification framework developed for the conditional BEQ instruction. Because branch instructions manipulate control flow directly, they do not utilize a standard, flat 31-instruction block, nor do they rely on a trailing JAL command to enforce the infinite loop structure.

After initializing comparison operands, the firmware sequences both taken and untaken branch targets. To maintain a standard switching profile, the program interleaves matching and mismatching register state evaluations, while varying the branch displacement distances across the memory space.

Listing Table 4.7 details the assembly layout deployed to characterize the unconditional JAL instruction. Unconditional jump operations including the register-indirect $JALR$ instruction rely on a control-flow architecture similar to the *B-Type* routines. However, they differ in that every individual jump instruction is structurally guaranteed to execute a control-flow transition to a new address space within the memory block.

To calculate the precise energy consumption per instruction, the raw current

Table 4.6: Firmware characterization loop for the BEQ (*B-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00100093	`addi x1, x0, 1`	Add Imm 1 to x0; store result in register x1.
00000113	`addi x2, x0, 0`	Load Imm value 0 into register x2 (li pseudo-op).
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
F81086E3	`beq x1, x1, -116`	Branch back to PC - 116 if register x1 equals x1.

metrics are processed using a weighted average based on the exact execution frequencies of each instruction inside the infinite loop. The energy metrics of auxiliary instructions (such as loop-control branches or address-seeding operations) are isolated and subtracted using the profiles compiled during prior characterization passes. This sequential extraction framework delivers highly accurate, isolated power metrics for every component of the implemented ISA.

Table 4.7: Firmware characterization loop for the *JAL* (*J-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
0240006F	jal x0, 36	Unconditional relative jump forward to PC + 36.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
0280006F	jal x0, 40	Unconditional relative jump forward to PC + 40.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
0280006F	jal x0, 40	Unconditional relative jump forward to PC + 40.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
F8DFF06F	jal x0, -116	Unconditional relative jump backward to PC - 116.
00000000	nop	No operation.
00000000	nop	No operation.

Chapter 5

Experimental Laboratory Setup for Hardware Characterization

This chapter presents the experimental laboratory setup and measurement infrastructure deployed to execute the physical post-silicon verification and electrical characterization of the fabricated Microprocessor.

By leveraging the pre-silicon *Vector-Based* simulation profiles alongside the on-chip *Debug-Module*, a direct cross-correlation can be established between the virtual layout models and the physical measurements. The experimental testing framework is structured to preserve methodological continuity with the Electronic Design Automation (EDA) environment, executing the identical testbenches and firmware payloads used during the pre-tapeout validation phases.

Physical verification is conducted by tracking internal memory states and pipeline *Logical Endpoints* at every discrete clock cycle using the integrated *Debug-Module*. Furthermore, the implementation of infinite execution loops within the characterization firmware substantially eases the time-domain constraints associated with post-silicon instrumentation tracking.

Operating at a nominal clock frequency of 100 MHz complicates post-silicon measurements, as it precludes real-time logging of current consumption using standard benchtop instruments. Resolving such high-speed fluctuations typically demands complex, high-bandwidth automated test equipment setups synchronized directly with the hardware test vectors. This approach would require significant modifications to the baseline testbenches, decoupling the physical testing pipeline from the pre-silicon simulation models.

By utilizing infinite execution loops engineered to maintain a stable, time-averaged *Switching-Activity*, the measurement window is no longer restricted by instrumentation sampling rates or tight timing triggers. This abstraction simplifies the post-silicon characterization flow, enabling highly accurate power profiling using standard laboratory instruments.

Figure 5.1 illustrates the structural schematic of the experimental test fixture utilized for hardware validation. Figure 5.2 shows the physical laboratory

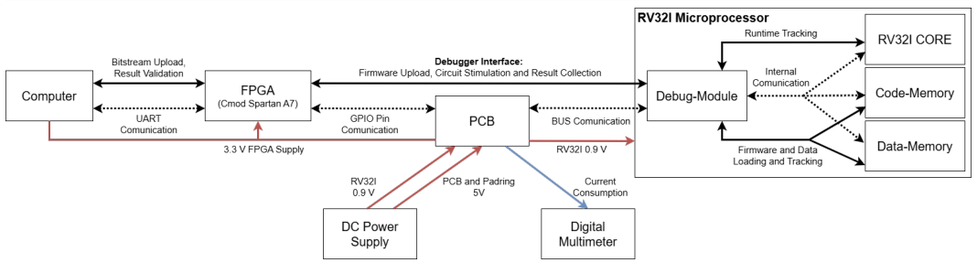

Figure 5.1: Structural block diagram of the post-silicon experimental validation framework, illustrating the connectivity between the *Computer*, FPGA bridge, custom application PCB, and laboratory instrumentation.

testbench configured for silicon characterization, and Figure 5.3 provides a detailed top-down view of the custom printed circuit board (PCB) assembly developed for this test vehicle.

As delineated in the architectural block diagram (Figure 5.1), the experimental setup integrates the following core subsystems:

- *Computer*: Drives the verification environment and processes data logs.
- **FPGA Bridge Module (Digilent Cmod Spartan A7):** Acts as the primary hardware interface and test controller.
- **Custom Application PCB:** Integrates the support circuitry, interconnect routing, and device sockets.
- **Precision *Power Supply*:** Delivers isolated, low-noise voltage rails to the system components.
- ***Digital Multimeter*:** Captures high-resolution current dissipation metrics.
- **RV32I Microprocessor:** The physical Device Under Test (DUT).

The *Computer* programs the FPGA bridge module and interfaces with it via a standard Universal Asynchronous Receiver-Transmitter (UART) protocol to stream firmware binaries and issue hardware control commands. The FPGA module bridges the *Computer* and the on-chip *Debug-Module*, routing signals through dedicated digital General-Purpose Input/Output (GPIO) pins.

During test execution, the FPGA systematically sequences the DUT control lines to upload and execute the compiled binaries. This configuration replicates the exact stimulus profile applied within the pre-silicon EDA environment, enabling cycle-by-cycle inspection of the internal register states and pipeline execution path.

The custom PCB (Figure 5.3) hosts the FPGA bridge module and routes the high-speed digital buses to the Microprocessor socket. To bridge the different signaling domains, the PCB integrates bidirectional voltage *Level-Shifters* that

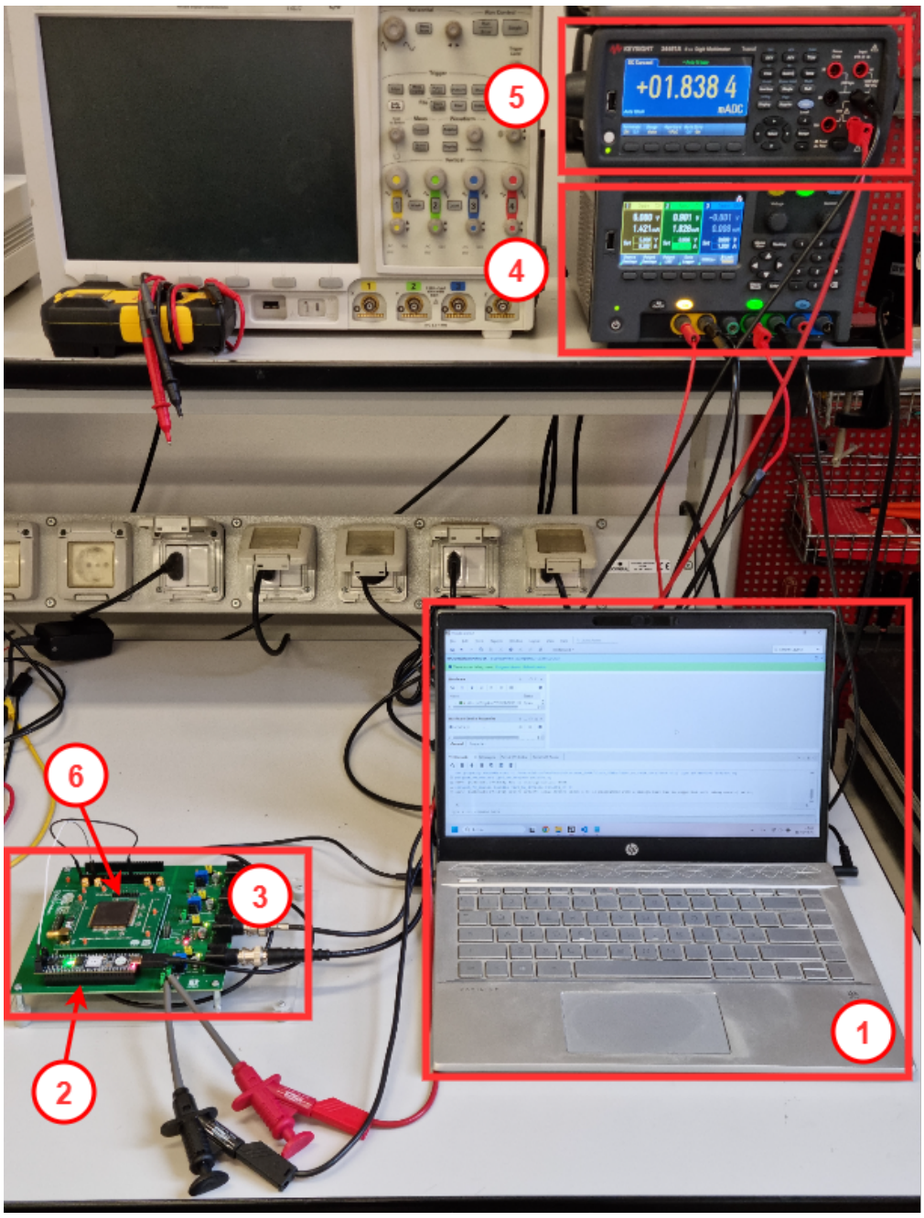

Figure 5.2: Experimental Measurement and Characterization Setup. 1) *Computer* 2) FPGA (Cmod Spartan A7) 3) Custom PCB 4) DC *Power Supply* 5) *Digital Multimeter* 6) RV32I Microprocessor (DUT - Device Under Test).

translate between the 3.3 V logic threshold of the FPGA and the nominal 1.8 V I/O ring voltage of the Microprocessor.

The precision *Power Supply* provides independent, isolated power rails, supplying 5.0 V to the peripheral PCB components and a nominal 0.9 V supply directly to the Microprocessor core. Concurrently, a high-resolution *Digital Multimeter* is placed in series with the core power rail to monitor power dissipation during steady-state loop execution. By cycling through the optimized assembly routines, this infrastructure measures the isolated average current consumption for each instruction within the Reduced Instruction Set Computer - Five (RISC-V) Instruction Set Architecture (ISA), providing a deterministic energy model of the physical Microprocessor.

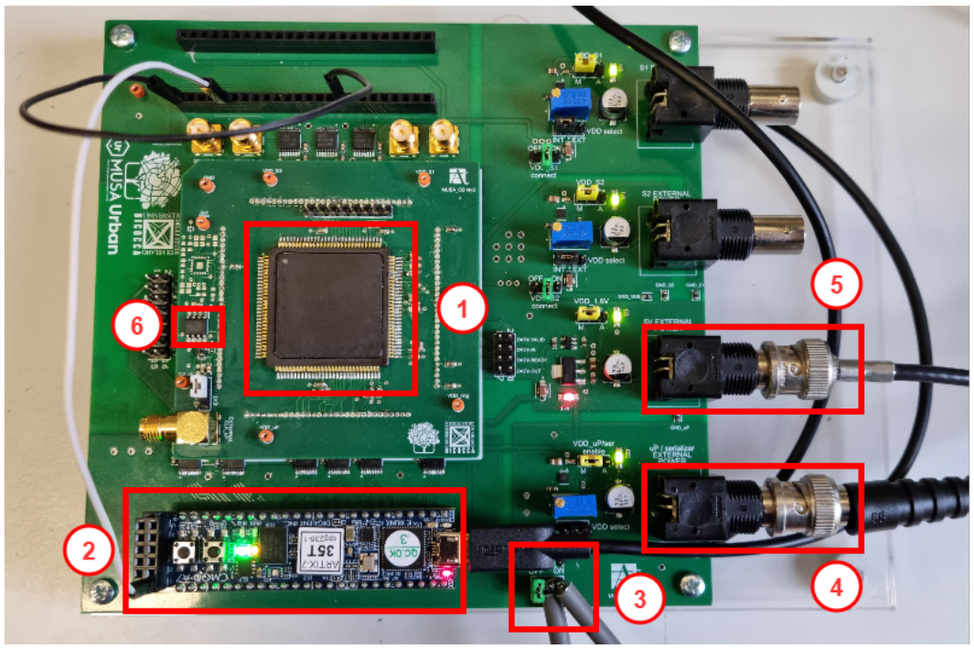

Figure 5.3: PCB Detail of Experimental Measurement and Characterization Setup. 1) RV32I Microprocessor (DUT - Device Under Test) 2) FPGA (Cmod Spartan A7) 3) *Digital Multimeter* Probe 4) RV32I Microprocessor Power Supply 5) PCB auxiliary and Padring Power Supply 6) PLL (Phase-Locked-Loop).

5.1 Printed Circuit Board Architecture and Design Strategy

This subsection details the hardware architecture and engineering specifications of the custom-designed printed circuit board (PCB) infrastructure, as depicted by the photograph of the assembled board in Figure 5.3 and structurally detailed in the circuit schematic of Figure 5.4.

The application PCB was engineered with a dual-purpose design mandate: to facilitate high-resolution post-silicon characterization in the laboratory, and to meet the stringent testing requirements of beam radiation experiments for aerospace emulation. Furthermore, structural versatility was integrated as a primary design parameter to maximize the reusability of the hardware platform across future iterative tape-outs of the Microprocessor architecture.

To satisfy these distinct validation paradigms, a modular split-board architecture was implemented, bifurcating the hardware platform into a standalone *Motherboard* and an interchangeable *Daughterboard*. This structural isolation is critical for upcoming radiation and Single-Event Effects (SEE) testing within beam-line facilities.

To ensure unattenuated beam penetration during radiation exposure, a strict keep-out zone was enforced directly beneath the integrated circuit package footprint. No signal routing, power planes, or copper traces were implemented within this localized vertical column on either board layer, preventing physical

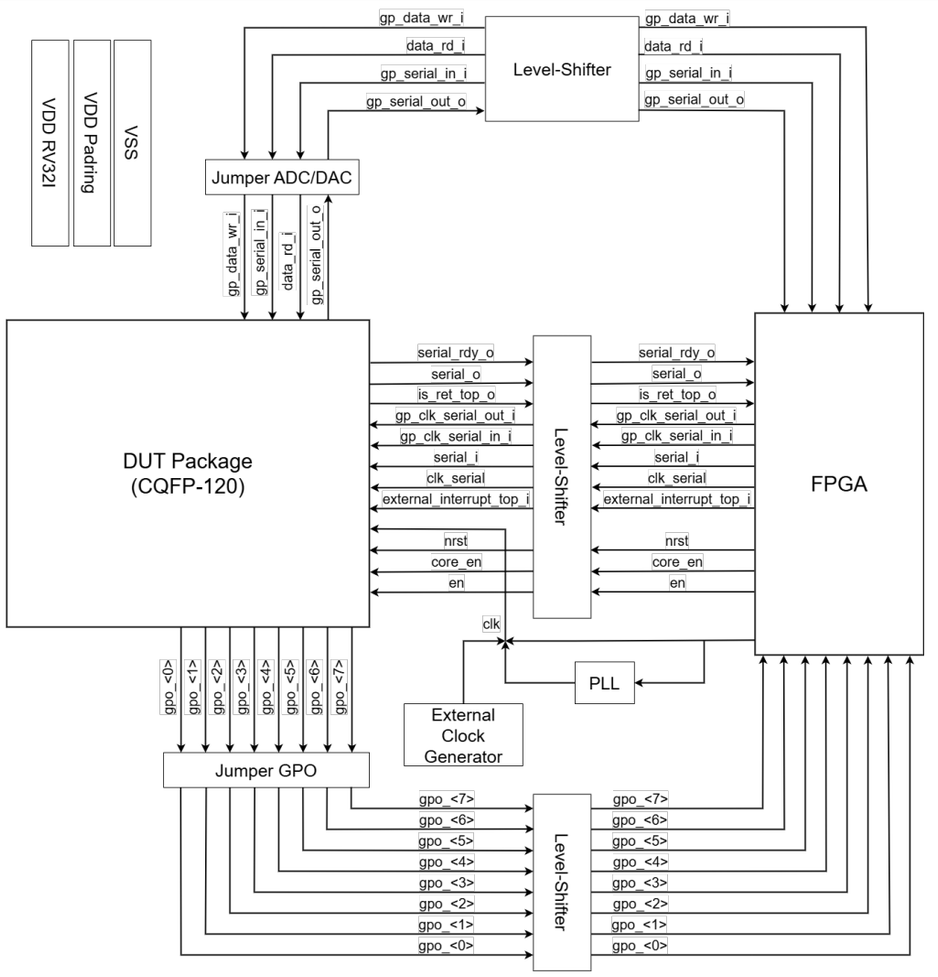

Figure 5.4: Schematic routing blueprint of the custom PCB, highlighting the bidirectional communication signal paths, level-shifting interfaces, and debug buses interconnecting the FPGA controller and the DUT.

metallization from scattering or absorbing the incident particle flux.

Furthermore, the PCB has been architected for reusability across diverse Microprocessor implementations, provided the interface remains compatible. This modularity is instrumental for evaluating future iterations and variations of the Microprocessor architecture.

From an electrical characterization standpoint, the PCB power tree was architected to fully isolate the primary *Core Power Supply* rail from the *Auxiliary Padring (I/O)* and peripheral board service rails. It also incorporates a dedicated power distribution path to supply the FPGA bridge module directly.

To achieve this level of operational flexibility, the platform supports multiple power provisioning modes. The board can be driven entirely by a single global voltage interface; in this configuration, an array of Onboard Low-Dropout (LDO) linear regulators segment and isolate individual supply channels, preventing cross-coupling noise and adapting the platform to varying benchtop measurement constraints.

Alternatively, power sequencing can be managed manually or automated dynamically via the FPGA bridge module. This capability allows precise

runtime control over the power-up and power-down sequencing of independent chip sub-blocks, a critical requirement for orchestrating automated laboratory testing and diagnosing latch-up conditions.

Furthermore, an array of accessible test points was integrated into the layout to facilitate signal debugging and hardware state inspection. Specifically, two high-precision, low-insertion-loss test points were placed in series with the core and I/O supply paths, enabling a standard *Digital Multimeter* to accurately sample continuous current consumption profiles during steady-state loop execution.

Given the 100 MHz target operating frequency of the Microprocessor, ensuring the spectral purity and structural integrity of the master clock signal was critical. To isolate the clock generation network from digital switching noise, a dedicated Phase-Locked Loop (PLL) synthesizer was integrated directly onto the PCB. The PLL acts as a clock cleaner and frequency multiplier, receiving a relaxed 50 MHz *Clock Reference* from the FPGA bridge module.

To maximize versatility, the clock distribution network on the PCB can be configured via physical jumpers to feed the DUT from three orthogonal sources: directly from the onboard PLL, via an *External Clock Generator* or directly from a dedicated FPGA hardware clock management tile.

The physical placement of the PLL chip was optimized to position it as close as geometrically possible to the Microprocessor clock input pin, minimizing trace inductance, impedance mismatches, and electromagnetic interference. Beyond regenerating and filtering phase noise from the raw input *Clock Reference*, the internal divider registers of the PLL can be dynamically programmed. This architectural feature allows the board to ingest lower-frequency, easily routable signals from the FPGA and synthesize highly stable, high-frequency, low-jitter clock waveforms directly at the boundary of the physical silicon.

Chapter 6

Experimental Results and State-of-the-Art Comparison

This chapter presents and evaluates the quantitative results obtained from both pre-tapeout Electronic Design Automation (EDA) physical simulations and post-silicon experimental laboratory measurements, focusing specifically on dynamic current dissipation vectors and worst-case voltage degradation (*IR-Drop*) profiles. A comparative analysis is subsequently established to benchmark the performance metrics of the fabricated core against state-of-the-art Reduced Instruction Set Computer - Five (RISC-V) Microprocessors reported in recent literature.

The extracted performance figures demonstrate the successful physical design and monolithic integration of an RV32I compliant RISC-V Microprocessor fabricated in a 28 nm Bulk Complementary Metal-Oxide-Semiconductor (CMOS) process node. This target implementation was achieved through the engineering of a customized digital physical implementation flow optimized to mitigate the heightened leakage current components intrinsic to standard Bulk CMOS technologies at advanced nodes.

Furthermore, the physical *Synthesis* and *Place&Route* constraints were specifically architected to withstand the radiation hazards characteristic of the target aerospace operating environment, hardening the physical fabric by systematically minimizing the peak *IR-Drop* across the localized power distribution networks.

To achieve this level of performance verification, a comprehensive gate-level *Signoff* and *Power-Analysis* simulation loop was executed at the back end of the digital implementation flow. When coupled with the architectural integration of the dedicated on-chip *Debug-Module*, this framework enables the execution of identical verification testbenches across both software-defined EDA simulators and hardware-in-the-loop laboratory instruments.

Methodological continuity is maintained by an experimental test fixture designed to correlate virtual simulation models directly with physical post-silicon measurements. Targeted assembly firmware payloads were synthesized to isolate

the average energy consumption of discrete instructions, providing an empirical foundation to characterize the total power envelope of the Microprocessor.

To ensure an equitable benchmarking comparison against alternative RISC-V hardware implementations, the underlying physical metrics must be translated into normalized Figures of Merit (FoM) that abstract, as far as practically possible, the specific characteristics of the chosen manufacturing process node and operating frequency.

First, the structural density of the Microprocessor core is quantified using Gate Equivalents (GE). This normalized metric defines the physical area of the circuit in terms of a standard reference logic gate specifically, the layout area of the smallest available minimum-sized inverter cell native to the chosen technology library. Expressing silicon area strictly in square millimeters (mm^2) introduces massive scaling skew due to the varying lithographic pitches of different technology nodes; utilizing GE resolves this variability, providing a technology-independent measure of Microprocessor architecture complexity.

Second, the dynamic power consumption is normalized to decouple the raw current metrics, traditionally recorded in milliamperes (mA), from the physical clock frequency of the system under test. Because dynamic power scales linearly with frequency, evaluating the core consumption in microwatts per megahertz (μW/MHz) normalizes the clock parameter to a unitary value, isolating the fundamental architectural energy efficiency of the execution path. It is worth noting that, because the proposed architecture is strictly single-cycle with a baseline Cycles Per Instruction (CPI) of 1, the power-to-frequency ratio (μW/MHz) is mathematically equivalent to pJ/cycle, and consequently, to pJ/instruction. To streamline the text and facilitate a technology-independent comparison with state-of-the-art Microprocessors, pJ/instruction (or simply pJ) is adopted as the unified energy metric throughout the remainder of this manuscript.

6.1 *Vector-Less* Statistical Analyses

This subsection evaluates the post-layout results obtained from static, probability-driven *Vector-Less* simulation passes. These analyses leverage uniform statistical parameters and localized *Switching-Activity* models to verify the structural robustness of the physical design while quantifying its core power consumption.

The *Vector-Less* simulation framework is initialized by configuring a global *Toggle-Rate* of 0.5 and an *Increase Switching Consumption* scaling factor of 0.3 at the inception of the *Place&Route* phase. The *Toggle-Rate* dictates the percentage of net transitions per clock cycle throughout the entire Microprocessor fabric, whereas the *Increase Switching Consumption* parameter adds an explicit dynamic overhead penalty to state transitions ($0 \rightarrow 1$ and $1 \rightarrow 0$) for every synthesized standard logic gate (switching cost multiplier).

The operational coefficients selected for this analysis represent a heavily guard-banded, worst-case corner for the proposed processing core. Statistically, this setup assumes that 50 % of the entire Microprocessor logic transitions concurrently at every single clock cycle. For a standard scalar RISC-V implementation, this operational profile significantly exceeds realistic instruction execution activity.

The primary objective of this deliberate over-stressing is to validate the structural integrity of the layout, with particular emphasis on the *Power-Grid* reliability under aerospace environment constraints. These highly conservative activity factors severely stress the physical rails, inducing maximum current density spikes and accentuating localized voltage drops (*IR-Drop*) to expose latent Microprocessor architecture layout bottlenecks.

To maximize simulation accuracy and capture timing cell dependencies, these probabilistic assumptions were tightly coupled with foundational switching profiles extracted from preliminary Toggle Count Format (*.TCF*) execution databases.

Table 6.1: *Vector-Less* analyses summary parameters and results.

Parameters and Results	**Value**
Technology Node	28 nm CMOS Bulk HPC+ TSMC
***Switching Activity* (Each Clock Cycle)**	50 %
Increase Switching Consumption	30 %
Static Current Consumption	0.012 mA
Dynamic Current Consumption	2.7 mA
Total Current Consumption	2.712 mA
Maximum *IR-Drop*	39 mV
Static-to-Dynamic Current Contribution	0.44 %
***Vector-Less* Analyses Corner**	TT, 25 °C, 0.9 V

Table 6.1 summarizes the *Signoff* electrical and technology metrics extracted from the *Vector-Less* analysis. Under these highly conservative toggle conditions, the core exhibits a static leakage current of 0.012 mA, a dynamic switching current of 2.7 mA at the nominal 0.9 V supply rail, and a peak localized *IR-Drop* of 39 mV.

The static leakage current contribution accounts for a mere 0.44 % of the total power dissipation envelope. This figure is significantly lower than the standard 10 % leakage threshold traditionally expected for advanced Bulk CMOS geometries and reported across standard baseline literature [82–84]. This reduction validates the low-power physical synthesis constraints and the multi-threshold voltage cell allocation deployed within the digital physical implementation flow to mitigate leakage current consumption.

The peak *IR-Drop* converges at 4.33 % of the nominal core supply voltage, well within the conservative 100 mV design margin established for *Signoff* closure. This low voltage-drop signature provides an exceptional electrical

safety margin against noise-induced bit-flips or localized delay degradation.

In radiation aerospace applications, Single Event Transient (SET) can dump structural charge track currents directly into digital nodes. By preserving a low baseline *IR-Drop*, the physical cells maintain an internal noise margin of more than 250 mV of aggregate voltage tolerance before encountering functional logic state degradation, successfully absorbing the combined effects of radiation-induced transient voltage swings and dynamic rail sagging.

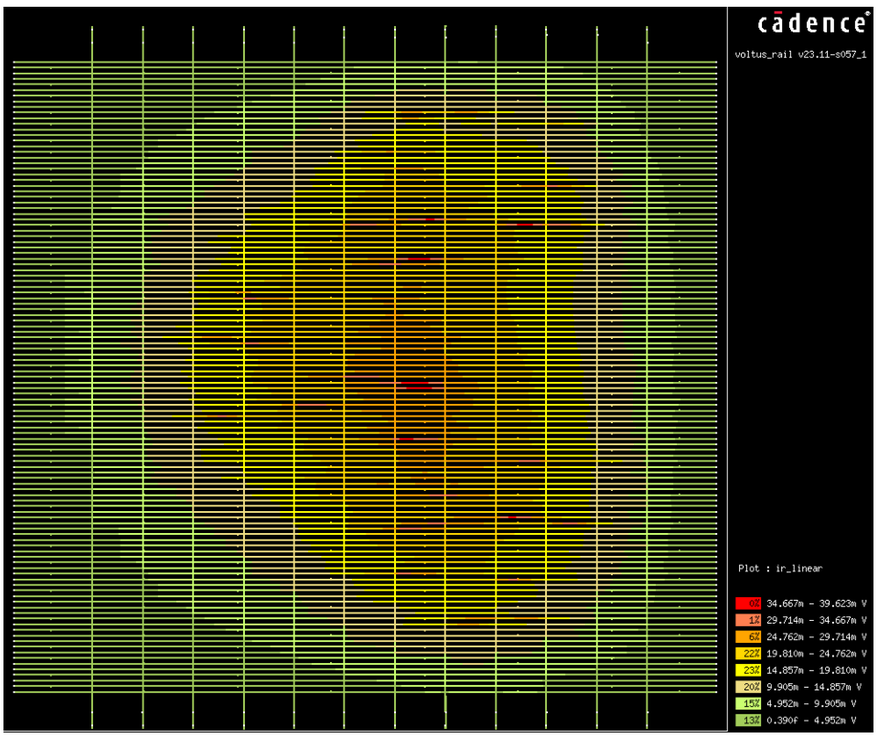

Figure 6.1: Post-layout 2D spatial voltage drop map of the Microprocessor, illustrating localized *IR-Drop* profiles under worst-case *Vector-Less* simulation.

Figure 6.1 illustrates the two-dimensional spatial distribution map of the Microprocessor core voltage degradation, extracted directly from the physical layout network via the EDA *Power-Grid Signoff* analyzer.

The color-gradient map demonstrates exceptional voltage uniformity across the physical die area, with no critical localized hot-spots or current-starved regions. This indicates that the layout grid structure provides low-impedance current paths to every standard cell row, ensuring high structural robustness under peak current demands without requiring physical layout adjustments or *Stripe* reinforcements.

Even under artificial worst-case switching conditions, the extracted static current metrics and localized *IR-Drop* margins confirm the effectiveness of the physical synthesis and power-distribution strategies integrated within the customized digital flow.

6.2 *Vector-Based* Dynamic Analyses

This subsection presents the post-layout validation results obtained from dynamic, deterministic *Vector-Based* simulations. These analyses leverage compile-time assembly firmware payloads to extract cell-level power vectors and isolate the exact energy consumption signature of each instruction within the RISC-V Instruction Set Architecture (ISA).

Table 6.2: *Vector-Based* analyses summary results.

Parameters	Vector-Based Simulations	Units
Technology Node	28nm CMOS Bulk HPC+ TSMC	[-]
Clock Frequency	100	[**MHz**]
Corner	TT, 0.9 V, 25 °C	[-]
Static Current Consumptions	0.011	[**mA**]
Static-to-Dynamic (Average)	0.61	[%]
Dynamic Average Consumptions	1.776	[**mA**]
	15.988	**pJ/Instruction**
Loop-Firmware Consumptions	1.85	[**mA**]
	16.65	**pJ/Instruction**
Blank-Firmware Consumptions	1.621	[**mA**]
	14.589	**pJ/Instruction**

Table 6.2 details the baseline structural operating metrics extracted via the back-annotated *Signoff* engine. Table 6.3 provides the isolated instruction-level dynamic power and energy breakdown, mapping metrics in both milliamperes (mA) and normalized energy-per-instruction figures (pJ/Instruction).

The extracted static leakage current converges at 0.011 mA, representing a minimal 0.61 % fraction of the active dynamic power envelope. This consistently suppressed leakage baseline confirms the efficacy of the leakage-mitigation constraints enforced during physical synthesis within the digital flow.

Table 6.2 also maps the power profiles of the control framework, isolating the *Blank-Firmware* execution current (1.621 mA) to capture the baseline pipeline clock distribution overhead, and the *Loop-Firmware* branch current (1.85 mA) to quantify the *JAL Loop-Back* contribution. By filtering out these baseline coefficients via a weighted algebraic average of the firmware instruction sequences, the localized energy cost of each specific ISA instruction is accurately extracted.

The instruction-level characterization matrix (Table 6.3) reveals that the subtraction operation (SUB) represents the peak dynamic power consumer at 2.161 mA (19.446 pJ). This peak is directly tied to the structural complexity of the look-ahead subtraction logic, which drives intense combinational gate toggling within the Arithmetic Logic Unit (ALU). Conversely, the conditional branch-if-equal instruction (BEQ) exhibits the lowest power profile within the active execution set, consuming 1.653 mA (14.88 pJ).

Cross-comparing the instruction formats yields distinct architecture insights into the internal switching dynamics of the Microprocessor. Specifically, when

Table 6.3: *Vector-Based* characterization of current consumption for instructions.

	Current Consumption	Dynamic Current	Current Calculated Weighted Sum	
Instructions/Units	**[mA]**	**[mA]**	**[mA]**	**pJ/Instruction**
LUI	1,859	1,848	1,848	16,629
AUIPC	1,846	1,834	1,834	16,505
JAL	1,776	1,764	1,764	15,880
JALR	1,771	1,760	1,760	15,840
BEQ	1,664	1,653	1,653	14,880
BNE	1,760	1,749	1,749	15,740
BLT	1,743	1,732	1,732	15,590
BGE	1,789	1,778	1,778	16,000
BLTU	1,757	1,746	1,746	15,710
BGEU	1,784	1,773	1,773	15,960
LB	1,808	1,797	1,789	16,097
LH	1,844	1,833	1,830	16,470
LW	1,837	1,826	1,821	16,391
LBU	1,772	1,761	1,748	15,735
LHU	1,790	1,779	1,768	15,916
SB	1,751	1,740	1,728	15,552
SH	1,762	1,751	1,740	15,659
SW	1,781	1,770	1,760	15,841
ADDI	1,974	1,963	1,967	17,703
SLTI	1,763	1,752	1,749	15,742
SLTIU	1,762	1,751	1,748	15,731
XORI	1,948	1,937	1,939	17,455
ORI	1,860	1,849	1,849	16,640
ANDI	1,800	1,789	1,787	16,082
SLLI	1,746	1,734	1,731	15,576
SRLI	1,751	1,740	1,736	15,628
SRAI	1,756	1,744	1,741	15,669
ADD	1,707	1,696	1,690	15,210
SUB	2,161	2,150	2,161	19,446
SLL	1,818	1,807	1,805	16,246
SLT	1,701	1,690	1,684	15,159
SLTU	1,701	1,690	1,684	15,159
XOR	1,699	1,688	1,682	15,138
SRL	1,818	1,807	1,805	16,246
SRA	1,739	1,728	1,723	15,511
OR	1,738	1,727	1,722	15,500
AND	1,719	1,708	1,703	15,324

evaluating register-immediate (*I-Type*) versus register-register (*R-Type*) operations, immediate variants systematically consume higher dynamic power than their purely register-based counterparts, as demonstrated by the fact that *ADDI* (1.967 mA, 17.703 pJ) exceeds *ADD* (1.69 mA, 15.21 pJ). This behavior occurs because processing arbitrary immediate operands increases the statistical variance and entropy of the second ALU input bus, thereby accentuating localized *Switching-Activity*.

Similarly, the memory scaling profiles show that memory access instructions scale predictably with the target bit-width of the transaction, establishing the strict hierarchies $LB < LH < LW$ and $SB < SH < SW$. This linear scaling directly reflects the capacitive charging overhead of activating progressively larger segments of the data bus and memory interfaces.

Finally, a clear distinction emerges when comparing signed versus unsigned operations. Unsigned load variants dissipate less power than signed operations, such that $LBU < LB$ and $LHU <$ LH, due to the complete deactivation of the sign-extension networks. Conversely, signed conditional branches such as *BLT* and *BGE* demonstrate reduced current draws compared to their unsigned counterparts, *BLTU* and *BGEU*. This reduction stems from the signed arithmetic logic, which averages fewer bit transitions across the high-order bits to

evaluate the branch conditions.

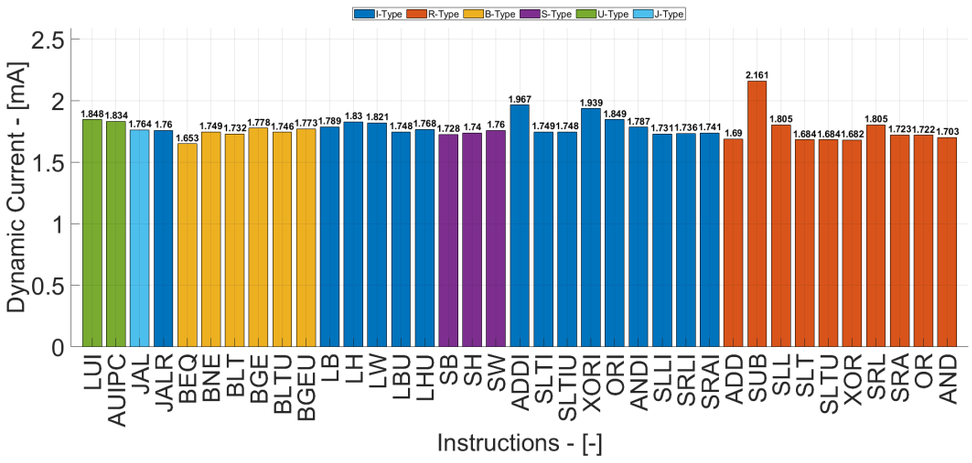

Figure 6.2: Simulated dynamic current consumption profiles across the RV32I ISA, categorized by RISC-V execution tiplogy.

Figure 6.2 provides a visual representation of the energy profiling across the RV32I ISA, sorting the individual instruction vectors by their standardized RISC-V functional classification.

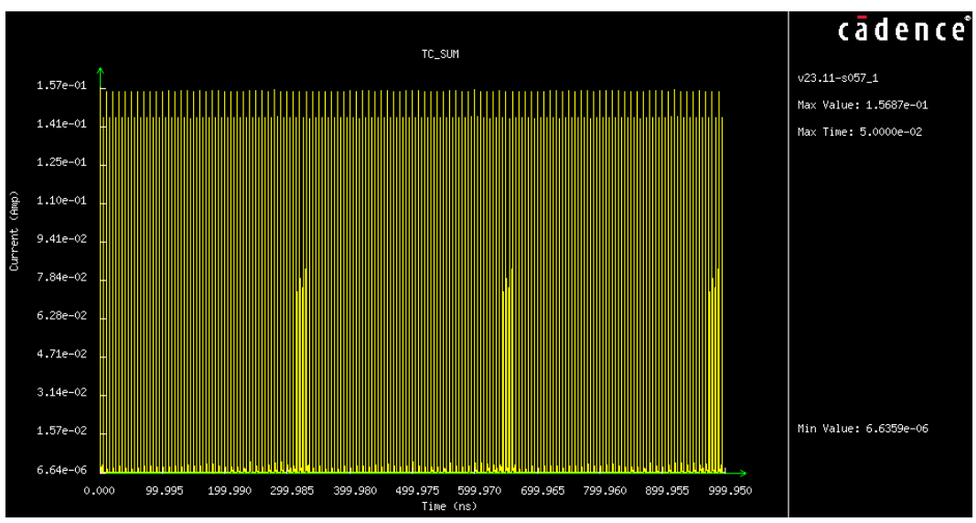

Figure 6.3: Simulated transient current profile over a 100-clock-cycle window, illustrating continuous clock-cycle power variations (*ADDI* instruction) and periodic peak modifications induced by the *Loop-Back* branch instruction.

Dynamic simulations also provide visibility into the temporal variations of core current consumption as a function of the executed software. Figure 6.3 charts the transient current waveform over a representative 100-cycle window during an *ADDI* validation pass.

A periodic modification in the current envelope occurs precisely every 31 clock cycles. This signature is an artifact of the unconditional *Loop-Back* branch instruction executed in the 32nd slot of the firmware structure. The lower peak amplitude of this periodic variance aligns with the metrics in Table 6.3,

confirming that the unconditional branch introduces less capacitive switching stress than the primary *ADDI* sequence.

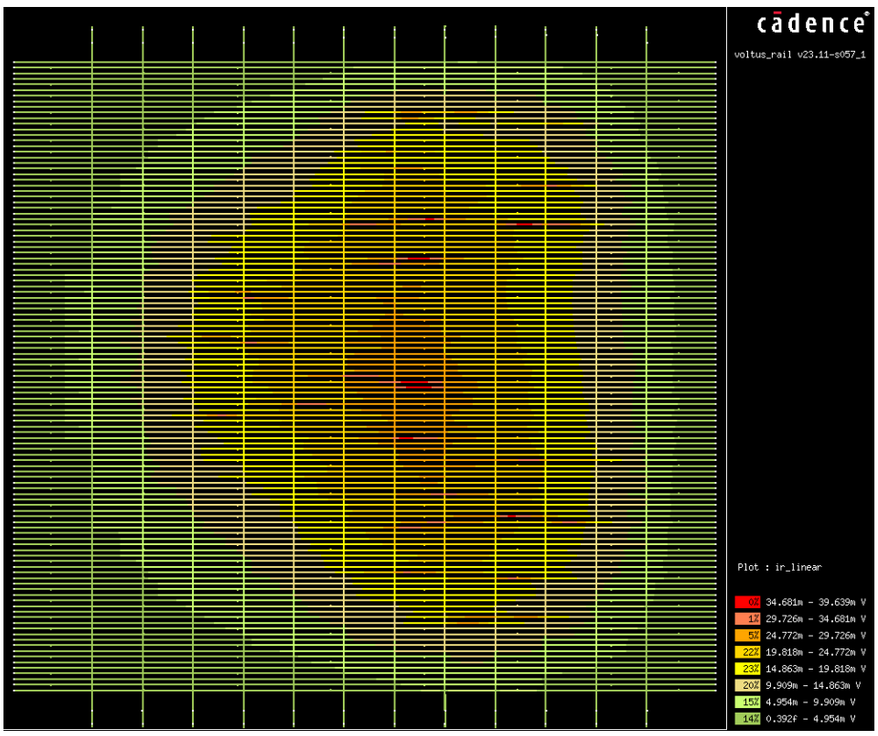

Figure 6.4: Post-layout dynamic *IR-Drop* spatial distribution map extracted during continuous execution of the *ADDI* characterization firmware.

Figure 6.4 depicts the post-layout voltage integrity analysis of the *PowerGrid* during active *ADDI* instruction loops. Under these physical execution conditions, the maximum dynamic *IR-Drop* peaks at a modest 34 mV, localized within less than 1 % of the integrated standard cell area. This tightly contained voltage drop confirms that the synthesized power grid mesh provides uniform low-resistance current distribution, ensuring structural reliability under continuous dynamic execution.

6.3 Experimental Validation and Power Characterization

This subsection details the quantitative results obtained from the physical laboratory testbench. By executing the dedicated characterization firmware payloads, post-silicon instruction-level power profiling was successfully conducted, demonstrating full functional correctness of the fabricated RV32I Microprocessor silicon implemented in a 28 nm Bulk CMOS HPC+ process node.

Table 6.4 summarizes the foundational operating parameters measured via the benchtop laboratory instrumentation. Table 6.5 outlines the post-silicon dynamic current consumption and normalized energy metrics extracted for

Table 6.4: Experimental laboratory setup analyses summary results.

Parameters	Laboratory Measurement	Units
Technology Node	28nm CMOS Bulk HPC+ TSMC	[-]
Clock Frequency	100	[**MHz**]
Corner	TT, 0.9 V, 25 °C	[-]
Static Current Consumptions	0.01 (0.053*)	[**mA**]
Static-to-Dynamic (Average)	0.55 (2.92*)	[%]
Dynamic Average Consumptions	1.81	[**mA**]
	16.287	**pJ/Instruction**
Loop-Firmware Consumptions	1.875	[**mA**]
	16.875	**pJ/Instruction**
Blank-Firmware Consumptions	1.664	[**mA**]
	14.976	**pJ/Instruction**

* Including the Consumptions Contribution of Auxiliary Circuitry

each individual instruction in the RV32I ISA.

During the physical characterization phase, the current baseline established by the *Blank-Firmware* execution serves as a diagnostic telemetry vector to verify instruction-stream decoding. If the physical current tracked by the digital multimeter matches or converges closely with the current consumption defined by *Blank-Firmware* (1.664 mA, 16.875pJ), it indicates that the Microprocessor is in an active state but is executing non-operational dummy fields (NOP routines). This baseline represents the minimum active hardware dissipation boundary, where the pipeline is toggling without shifting variable state weights through the ALU execution units.

The total static current recorded during benchtop testing is 0.053 mA, as shown in Table 6.4. This value reflects the un-gated static leakage floor of the entire silicon die, which hosts secondary auxiliary macros alongside the primary processing core. Although these Intellectual Property (IP) blocks were kept inactive during the Microprocessor test vectors, they still exhibit an irreducible baseline leakage current.

To isolate the specific footprint of the Microprocessor, the simulated leakage contribution of these auxiliary modules which sums to 0.043 mA was subtracted as a background coefficient from the global empirical silicon metrics. Through this differential analysis, the net static leakage current of the Microprocessor core was indirectly isolated at approximately 0.010 mA. This physical measurement closely correlates with the pre-silicon *Signoff* estimation of 0.011 mA, providing empirical proof of the leakage-reduction mechanisms implemented during the physical design phase.

The post-silicon electrical characterization (Table 6.5) closely reflects the relative instruction-level energy variations observed during gate-level simulation passes. The physical execution models confirm the structural trends discussed in the pre-silicon analysis, maintaining identical power hierarchies across all instruction types and bit-width configurations.

Figure 6.5 provides the final empirical histogram of the physical core current

Table 6.5: Experimental laboratory setup characterization of current consumption for instructions.

	Current Consumption	Current Measured	Current Calculated Weighted Sum	
Instructions/Units	**[mA]**	**[mA]**	**[mA]**	**pJ/Instruction**
LUI	1,941	1,888	1,888	16,996
AUIPC	1,862	1,809	1,807	16,262
JAL	1,860	1,807	1,807	16,263
JALR	1,787	1,734	1,734	15,606
BEQ	1,747	1,694	1,694	15,246
BNE	1,864	1,811	1,811	16,299
BLT	1,857	1,804	1,804	16,236
BGE	1,882	1,829	1,829	16,461
BLTU	1,869	1,816	1,816	16,344
BGEU	1,908	1,855	1,855	16,695
LB	1,887	1,834	1,827	16,443
LH	1,896	1,843	1,837	16,534
LW	1,905	1,852	1,847	16,626
LBU	1,850	1,797	1,785	16,066
LHU	1,866	1,813	1,803	16,229
SB	1,820	1,767	1,755	15,791
SH	1,834	1,781	1,770	15,926
SW	1,846	1,793	1,782	16,042
ADDI	2,055	2,002	2,006	18,055
SLTI	1,835	1,782	1,779	16,011
SLTIU	1,834	1,781	1,778	16,002
XORI	1,992	1,939	1,941	17,470
ORI	1,925	1,872	1,872	16,847
ANDI	1,869	1,816	1,814	16,327
SLLI	1,835	1,782	1,779	16,011
SRLI	1,842	1,789	1,786	16,076
SRAI	1,842	1,789	1,786	16,076
ADD	1,777	1,724	1,719	15,467
SUB	2,181	2,128	2,137	19,233
SLL	1,898	1,845	1,844	16,595
SLT	1,780	1,727	1,722	15,495
SLTU	1,780	1,727	1,722	15,495
XOR	1,777	1,724	1,719	15,467
SRL	1,898	1,845	1,844	16,595
SRA	1,816	1,763	1,759	15,831
OR	1,816	1,763	1,759	15,831
AND	1,800	1,747	1,742	15,682

consumption per instruction, with individual bars color-coded according to their architectural RISC-V execution class.

6.4 Simulation Result vs. Post-Silicon Experimental Measurement

This subsection establishes a rigorous cross-correlation between the pre-tapeout EDA power-integrity simulation models and the empirical data harvested from the post-silicon laboratory test fixture. Evaluating the delta ($|\Delta|$) between these two domains qualifies the mathematical accuracy of the *Signoff* models and validates the predictive capability of the power analysis flow.

To minimize environmental and process skew during the correlation loop, the *Signoff* gate-level simulations were executed utilizing the *Typical-Typical* (TT) foundry device models under a controlled nominal junction temperature of 25°C and a 0.9 V supply rail. Mirroring these conditions, the physical post-silicon testing was performed within an automated benchtop laboratory environment strictly regulated at an ambient temperature of 25°C, with the custom power distribution network delivering an identical nominal 0.9 V bias

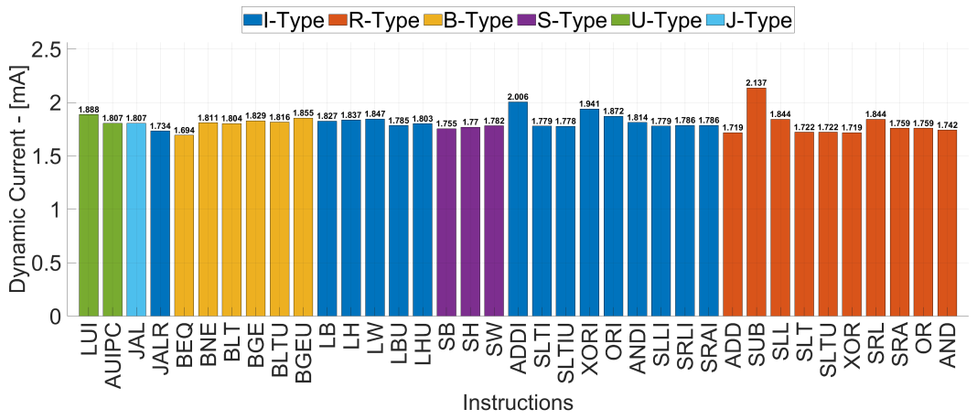

Figure 6.5: Experimental laboratory setup dynamic current consumption profiles across the RV32I ISA, categorized by RISC-V execution tiplogy.

Table 6.6: Comparison between simulated result and experimentally measurement of the proposed Microprocessor.

Parameters	**Simulations**	**Laboratory Measurement**	**Units**
Technology Node	28 nm CMOS Bulk HPC+ TSMC		[-]
Clock Frequency	100		**[MHz]**
Corner	TT, 0.9 V. 25 °C		[-]
Static Current Consumptions	0.011	0.01 (0.053*)	**[mA]**
Static-to-Dynamic (Average)	0.61	0.55 (2.92*)	[%]
Dynamic Average Consumptions	1.776	1.81	[mA]
	15.988	16.287	**pJ/Instruction**
Loop-Firmware Consumptions	1.850	1.875	[mA]
	16.65	16.875	**pJ/Instruction**
Blank-Firmware Consumptions	1.621	1.664	[mA]
	14.589	14.976	**pJ/Instruction**
Average |Δ|	0.037		[mA]
	0.337		**pJ/Instruction**
|Δ|-to-Dynamic (Average)	2.08	2.04	[%]

* Including the Consumptions Contribution of Auxiliary Circuitry

to the Microprocessor core (Table 6.6).

Table 6.6 summarizes the primary baseline operating metrics recorded across both domains. The net static core leakage current converges at 0.011 mA in simulation and 0.01 mA via the indirect experimental extraction method (following the systematic de-embedding of co-integrated macro components).

Expressing these metrics relative to the overall core behavior, the pre-silicon static leakage current represents 0.61 % of the mean simulated instruction consumption (1.776 mA, corresponding to a normalized value of 15.988 pJ). In the physical domain, the experimental static current constitutes 0.55 % of the measured core active profile.

A closer look at the active execution profiles demonstrates a mean absolute discrepancy in dynamic current consumption of just 0.037 mA (0.337 pJ) between the virtual and physical domains. This delta translates to a relative error margin of 2.04 % with respect to the empirical laboratory metrics, and

2.08 % relative to the back-annotated simulation baseline.

This tight alignment between the software-derived EDA models and the physical silicon validation proves that the *Signoff* layout modeling and *Power-Analyses* simulation frameworks capture the physical switching dynamics and parasitic profiles of the fabricated core with exceptional accuracy.

Table 6.7: Instruction-level dynamic power and energy correlation between simulations results and experimental laboratory setup measurements.

	Simulations		**Laboratory Measurement**		$\lvert\Delta\rvert$	
Instructions	**[mA]**	**pJ/Instruction**	**[mA]**	**pJ/Instruction**	**[mA]**	**pJ/Instruction**
LUI	1.848	16.629	1.888	16.996	0.041	0.366
AUIPC	1.834	16.505	1.807	16.262	0.027	0.244
JAL	1.764	15.880	1.807	16.263	0.043	0.383
JALR	1.760	15.840	1.734	15.606	0.026	0.234
BEQ	1.653	14.880	1.694	15.246	0.041	0.366
BNE	1.749	15.740	1.811	16.299	0.062	0.559
BLT	1.732	15.590	1.804	16.236	0.072	0.646
BGE	1.778	16.000	1.829	16.461	0.051	0.461
BLTU	1.746	15.710	1.816	16.344	0.070	0.634
BGEU	1.773	15.960	1.855	16.695	0.082	0.735
LB	1.789	16.097	1.827	16.443	0.038	0.346
LH	1.830	16.470	1.837	16.534	0.007	0.065
LW	1.821	16.391	1.847	16.626	0.026	0.235
LBU	1.748	15.735	1.785	16.066	0.037	0.331
LHU	1.768	15.916	1.803	16.229	0.035	0.313
SB	1.728	15.552	1.755	15.791	0.027	0.240
SH	1.740	15.659	1.770	15.926	0.030	0.268
SW	1.760	15.841	1.782	16.042	0.022	0.201
ADDI	1.967	17.703	2.006	18.055	0.039	0.352
SLTI	1.749	15.742	1.779	16.011	0.030	0.269
SLTIU	1.748	15.731	1.778	16.002	0.030	0.270
XORI	1.939	17.455	1.941	17.470	0.002	0.014
ORI	1.849	16.640	1.872	16.847	0.023	0.207
ANDI	1.787	16.082	1.814	16.327	0.027	0.245
SLLI	1.731	15.576	1.779	16.011	0.048	0.435
SRLI	1.736	15.628	1.786	16.076	0.050	0.448
SRAI	1.741	15.669	1.786	16.076	0.045	0.407
ADD	1.690	15.210	1.719	15.467	0.029	0.257
SUB	2.161	19.446	2.137	19.233	0.024	0.213
SLL	1.805	16.246	1.844	16.595	0.039	0.349
SLT	1.684	15.159	1.722	15.495	0.037	0.337
SLTU	1.684	15.159	1.722	15.495	0.037	0.337
XOR	1.682	15.138	1.719	15.467	0.037	0.330
SRL	1.805	16.246	1.844	16.595	0.039	0.349
SRA	1.723	15.511	1.759	15.831	0.036	0.320
OR	1.722	15.500	1.759	15.831	0.037	0.331
AND	1.703	15.324	1.742	15.682	0.040	0.358

Table 6.7 isolates the instruction-by-instruction correlation matrix across representative fields of the ISA. The dataset is cross-mapped in both raw current values (mA) and normalized energy metrics pJ/Instruction to abstract the specific clock frequency constraints and enable technology-independent verification. Because the energy expressed in pJ/Instruction eliminates time-domain scaling dependencies, it acts as a reliable metric to verify that the Microprocessor architecture switching weight of each instruction is modeled correctly prior to tape-out.

6.5 Microprocessor Performance Comparison with the State-of-the-Art

This subsection establishes a comprehensive performance benchmarking analysis, cross-comparing the extracted pre-silicon and post-silicon core metrics against comparable RISC-V hardware implementations found in recent literature.

Because the open-source RISC-V ISA does not dictate a binding Microprocessor architecture template, execution path, or physical layout framework, direct chip-to-chip comparisons are non-trivial. To circumvent these constraints, normalized Figures of Merit (FoM) specifically kilo-Gate Equivalents (kGE) for structural density and pJ per Instruction (μW/MHz) for energy efficiency are utilized to abstract away the specific variables of each fabrication technology.

As established, the GE metric normalizes physical core area to the geometry of a minimum-sized inverter cell native to each respective process node library. This abstraction filters out the major scaling distortion introduced by raw area footprints expressed in square millimeters (mm^2), which are heavily skewed by the lithographic pitch of the chosen node. Similarly, core power dissipation scales linearly with the target operational clock frequency. Expressing dynamic power in μW/MHz normalizes the frequency factor to unity. As previously discussed, this ratio translates directly to pJ/instruction. Normalizing consumption to pJ/instruction ensures a fair evaluation between single-cycle and pipelined architectures, effectively abstracting the operational frequency and isolating the architectural energy efficiency of the execution datapath. However, while kGE normalizes area across different technologies, it must be noted that cross-node absolute energy comparisons (e.g., 28 nm vs. 130/180 nm) remain intrinsically dominated by technology scaling.

Table 6.8: Microprocessor performance comparison with the State-of-the-Art.

Project Name	**This Work**	**Siwa [89]**	**Mriscv [90, 91]**	**Riscy [92]**	**Zero-Riscy [92]**	**Micro-Riscy [92]**	**NoX [93]**	**[94]**	**[95]**	**[96]**	**OpenRISC [97]**
Technology	28 nm CMOS Bulk TSMC	180 nm	130 nm GP TSMC	65 nm UMC	65 nm UMC	65 nm UMC	SKY130 nm	45 nm UMC	40 nm UMC	65 nm UMC	65 nm UMC
Frequency [MHz]	100	20	160	100	100	100	100	100	198.02	357	362
Area [mm^2]	0.01	0.672	0.12	0.703	0.326	0.2	0.295	0.22	0.36	0.068	0.064
Equivalent Gate [kGE]	13.649	-	-	40.7	18.9	11.6	-	-	-	46.9	44.5
Word Size [Bit]	32	32	32	32	32	16	32	32	32	32	32
ISA	RV32I	RV32I	RV32IM	RV32IMC	RV32IMC	RV32E(C)	RV32I-Zicsr	RV32I	RV32IM	RV32IM	RV32IM
Component	RISC-V Core	RISC-V Core + 8 kB SRAM	RISC-V Core	RISC-V Core	RISC-V Core	RISC-V Core	RISC-V Core	RISC-V Core	RISC-V Core	RISC-V Core	RISC Core
MCMM (Corner)	Yes	No	No	No	No	No	Yes	No	No	No	No
Average CPI	1 (single-cycle)	4	(Pipeline)	1.31	1.92	1.51	(Pipeline)	(Pipeline)	(Pipeline)	(Pipeline)	(Pipeline)
Process	TT	-	-	-	-	-	TT	-	-	-	-
Temperature [°C]	25	-	-	-	-	-	25	-	-	-	-
VDD [V]	0.9	1.8	1.2	1.2	1.2	1.2	1.8	1.2	1.2	1.08	1.2
pJ/Instruction	15.988 / 16.287*	48.31	167	41.57	25.57	12.54	231.2	21	17.36	26.28	33.8

* Simulations / De-embedded Laboratory Measurement

Evaluating the alternative Microprocessor architectures requires balancing several interdependent vectors, including structural layout complexity, energy efficiency, target operational frequency, instruction-set capabilities, and the physical integration of on-chip peripherals, memories, or custom Digital Signal Processing (DSP) accelerators.

The proposed Microprocessor core achieves a highly competitive design

space trade-off between architectural throughput and power consumption (Table 6.8). Critically, most state-of-the-art literary designs omit rigorous Multi-Corner Multi-Mode (MCMM) *Signoff* conditions during physical placement and synthesis, with the notable exception of Nox [93]. Neglecting MCMM constraints artificially inflates nominal performance figures and reduces gate counts, but yields a fragile physical fabric vulnerable to environmental or process drifts. In contrast, the robust physical flow designed for this core prioritizes reliability under aerospace radiation constraints. The fact that it achieves superior energy efficiency despite these strict structural safety margins demonstrates its Microprocessor architectural optimization.

Detailed cross-examinations against specific literary implementations reveal distinct insights when compared directly to the proposed core. The implemented core demonstrates superior metrics over the Siwa [89] architecture; although both execute the baseline RV32I ISA, Siwa relies on a multi-cycle execution scheme that degrades throughput, resulting in a Clocks Per Instruction (CPI) factor of 4 compared to the optimal single-cycle execution achieved in this work. A direct area comparison with Siwa is limited because that core is fabricated in a mature 180 nm node, reported strictly in absolute area, and includes a monolithic 8 kB SRAM macro; however, the proposed core achieves a higher maximum clock frequency of 100 MHz versus 20 MHz while simultaneously reducing power consumption from 48.31 pJ down to 15.98 pJ.

The proposed core also offers a significantly more efficient power profile than Mriscv [90, 91]. The Mriscv core includes the M extension for hardware multiplication and division, which expands its computational capacity and supports a higher top frequency of 160 MHz versus 100 MHz, but its power dissipation is profoundly higher at 167 pJ compared to the 15.98 pJ of this work. This extreme power penalty is not justified by the incremental throughput gain of the multiplication module, making the proposed core far more suitable for power-constrained applications.

When cross-compared with the Parallel Ultra-Low-Power (PULP) family, the integrated core demonstrates highly competitive trade-offs. Regarding Riscy [92], Zero-Riscy [92], and Micro-Riscy [92], the energy metrics reported in the referenced work are data estimates and projections, they do not represent direct simulation data. Even against these extrapolated baselines, the proposed Microprocessor remains highly competitive. The Riscy core is larger (40.7 kGE) and exhibits a higher energy consumption (41.57 pJ). Zero-Riscy maintains a higher computational capability than the proposed design by leveraging the M module; nevertheless, it remains larger (18.9 kGE) and demonstrates a higher energy footprint (25.57 pJ vs 15.98 pJ). Finally, Micro-Riscy is an Internet of Things (IoT)-oriented version that utilizes the reduced RV32E architecture (featuring 16 registers instead of 32) to aggressively minimize area (11.6 kGE) and lower the energy footprint to 12.54 pJ. While Micro-Riscy achieves a

lower absolute energy cost, the proposed RV32I implementation provides the full computational capacity of the standard ISA while maintaining a highly competitive energy profile.

Furthermore, the implemented core achieves superior efficiency compared to both Nox [93], which dissipates 231.2 pJ, and [94] which dissipates 21 pJ, while maintaining architectural parity across identical RV32I parameters.

Conversely, the [95] architecture represents an exceptionally high-performing implementation that achieves a significantly higher target frequency of 198.02 MHz and incorporates the M extension. Given that its normalized power overhead of 17.36 pJ is nearly equivalent to the 15.98 pJ of this work, [95] stands out as a structurally superior design for high-performance envelopes.

Finally, both [96] and OpenRISC [97] target aggressive operating frequencies of 357 MHz and 362 MHz, respectively. However, this performance target requires substantial circuit retiming and pipelining overhead, inflating their silicon area to 46.9 kGE and 44.5 kGE without the justification of an M extension. The core presented in this work proves superior for power-critical systems compared to them, offering a much more compact silicon footprint and a lower power metric against the 26.28 pJ of [96] and 33.8 pJ of OpenRISC.

Chapter 7

Auxiliary Radiation-Hardened Subsystems and Computational Extensions

This chapter introduces the design frameworks, exploratory layouts, and ongoing research pipelines for custom auxiliary subsystems designed to interface with the Microprocessor core. These macro blocks represent in-progress architectural developments required to transition the existing Microprocessor core into a full-scale, radiation-hardened (rad-hard) system-on-chip optimized for aerospace applications.

As established throughout this work, monolithic memory blocks represent highly critical vulnerabilities within high-reliability environments due to their extreme susceptibility to radiation-induced Single-Event Effects (SEE). Because the memory array establishes and tracks the state variables of the Microprocessor during software execution, an unmitigated bit-flip (such as a Single Event Transient (SET) or Single Event Upset (SEU)) can irreversibly corrupt the instruction stream, leading to catastrophic system lockups or inconsistent computational execution. To neutralize these failure modes, this ongoing research phase explores alternative topological and physical design methodologies aimed at hardening storage elements while maximizing cell packing density.

First, the current design and integration framework of a custom, Radiation-Hardened-by-Design (RHBD) Static Random-Access Memory (SRAM) cell is presented. Unlike traditional mitigation techniques, this topology avoids the power and area overhead of Triple Modular Redundancy (TMR) and majority-voting networks. Instead, SEU immunity is achieved natively by embedding a filtering capacitance directly within the cross-coupled inverter storage nodes of each cell. At the physical layout level, this extra capacitance is routed through the upper metal layers directly overlying the standard-cell active area, achieving hard-wired nodes without adding a physical area penalty to the base cell footprint.

Concurrently, parallel exploration is currently underway focusing on param-

eterizable macro arrays generated via a commercial foundry *SRAM-Compiler* provided by Taiwan Semiconductor Manufacturing Company (TSMC). This compiler leverages highly optimized, parametric layout libraries to rapidly instantiate diverse memory capacities and aspect ratios. While compiler driven macros dramatically compress the product development cycle, the next phase of this development necessitates rigorous beam characterization to verify their native radiation hardness for aerospace applications.

Beyond the storage subsystems, active work is being carried out on the integration of a high-performance Floating Point Unit (FPU) targeting a 500 MHz execution frequency. This hardware accelerator is designed to natively support the standard Reduced Instruction Set Computer - Five (RISC-V) M (integer multiplication and division) and F (single-precision floating-point arithmetic) Instruction Set Architecture (ISA) extensions. Implementing these specialized execution blocks expands the Microprocessor architecture throughput by offloading transcendental mathematics, square roots, and floating-point vectors from the core pipeline, these extensions expanding its application space.

Finally, this work presents the current development status and synthesis of the digital logic control architecture for a 640 MHz Phase-Locked Loop (PLL) tailored for aerospace environments. The clock distribution network represents the single most critical signaling infrastructure in any synchronous digital system, and is especially crucial for timing margin closures within high-frequency Microprocessors. Any degradation in clock integrity such as dynamic phase jitter or duty-cycle distortion can induce *Setup* or *Hold* violations, leading to unpredictable instruction decoding. Maintaining a clean, uniform reference clock is imperative, as even a robustly routed clock tree generated during the Clock Tree Synthesis (CTS) phase of the *Place&Route* flow cannot compensate for a noisy or unstable upstream clock source.

7.1 Radiation-Hardened-by-Design SRAM Architecture and Verification Methodology.

This subsection presents the design of a custom Static Random-Access Memory (SRAM) employing Rad-Hard-by-Design (RHBD) layout techniques. Specific emphasis is placed on the automated verification topology engineered to validate memory integrity, functional correctness, and reliability boundaries.

In aerospace electronic systems, memory architectures represent one of the most critical and SEU sensitive bottlenecks due to the high flux of ionizing radiation. Consequently, the storage subsystems within this research program have been decoupled from the primary Microprocessor core to undergo independent topological optimization.

Traditional structural mitigation techniques rely heavily on Triple Modular

Redundancy (TMR) combined with majority-voting logic. While effective, TMR introduces an unmitigated threefold penalty in silicon area and dynamic power dissipation. Given the inherent architecture complexity of modern Microprocessors, such an overhead can easily become prohibitive. To bypass these geometric and power constraints, this research introduces an embedded SRAM subsystem engineered to achieve native radiation hardness at the cell level, eliminating structural power and area penalties.

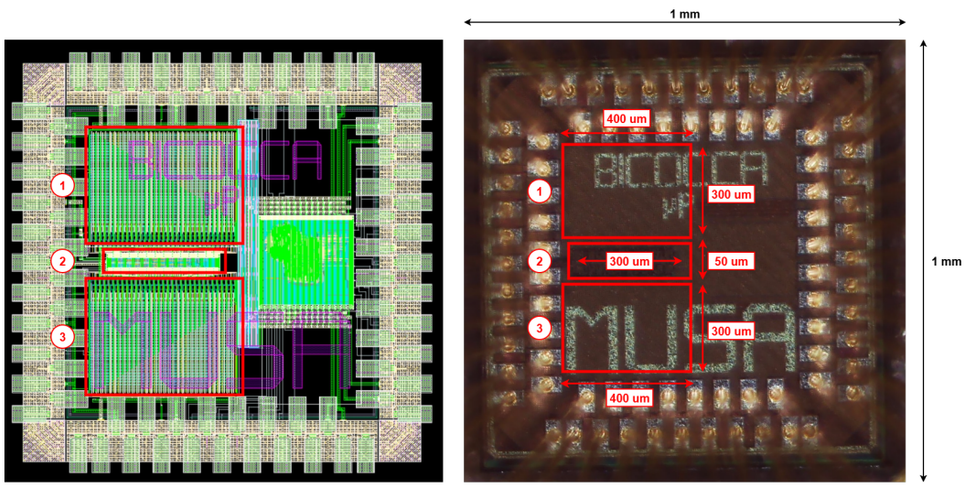

Figure 7.1: Top-level analog layout view showing the co-integrated standard and RHBD SRAM arrays (Left) alongside the central serializer; (Right) Die micrograph highlighting the manufactured silicon macro area. 1) SRAM Rad-Hard, 2) SRAM Digital Serializer and 3) SRAM NO Rad-Hard.

The developed physical prototype combines two distinct SRAM arrays and a shared 100 MHz serial interface co-integrated on a single silicon die utilizing a 28 nm Bulk Complementary Metal-Oxide-Semiconductor (CMOS) HPC+ process node. Figure 7.1 illustrates the complete physical layout view on the left, demonstrating the *Analog-on-Top* macro assembly of the dual arrays, alongside the fabricated die micrograph on the right.

To provide a rigid baseline for empirical cross-correlation under radiation beam testing, the layout features twin configurations: one array is synthesized using a standard low-power memory bit-cell architecture, while the parallel array implements custom RHBD structural modifications. This dual-macro topology enables concurrent, side-by-side radiation exposure, providing definitive empirical validation of the RHBD methodology as the standard array experiences soft errors (bit-flips) while the hardened macro maintains state integrity.

As illustrated in Figure 7.2, the RHBD storage mechanism integrates a localized decoupling capacitance directly inside the cross-coupled latch inverter nodes of each individual bit-cell. This embedded capacitive filter attenuates transient voltage spikes induced by ion strikes, preventing the node from exceeding the critical charge threshold required to flip the state [56, 59, 60].

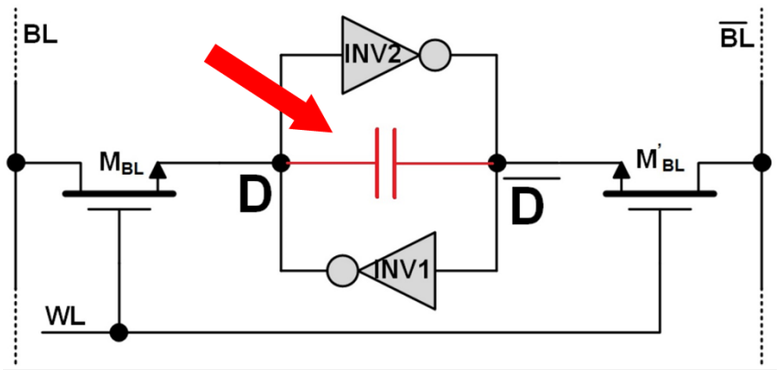

Figure 7.2: Schematic of the proposed radiation-hardened-by-design SRAM bit-cell featuring an integrated area-neutral decoupling capacitor.

To prevent this additional component from expanding the cell footprint, the capacitor is routed entirely through the upper metallization layers directly overlying the active silicon area of the underlying standard cell. This vertical layout integration completely eliminates physical area overheads and active leakage penalties relative to the baseline commercial cell.

The data path is managed by an on-chip serial interface operating at a nominal clock frequency of 100 MHz. This block arbitrates external communication with both SRAM arrays via an optimized bidirectional shift register. The shift register interfaces in parallel with the memory matrices, executing read or write cycles based on testbench commands stored in dedicated control register slots.

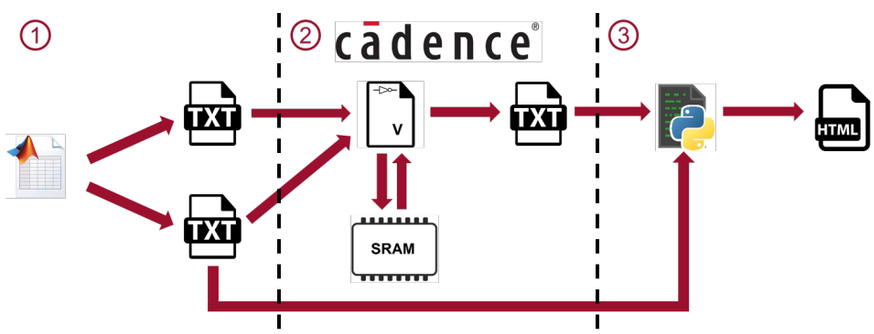

Figure 7.3: Block scheme of testbench methodology to verify the SRAM. 1) Matlab script 2) Cadence simulation 3) Graphical result analysis.

Memory validation requires rigorous stress-testing patterns engineered to provoke write-disturbance or read-destructive failure modes under tight timing margins. To standardize this process, an automated verification pipeline was designed and executed.

Figure 7.3 shows the structural block diagram of the verification environment. This architecture establishes a clear layer of abstraction between testbench generation and hardware stimulus, significantly streamlining laboratory characterization. This automation is particularly critical for data logging during irradiation test campaigns under ion beams, where real-time

error telemetry must be collected efficiently.

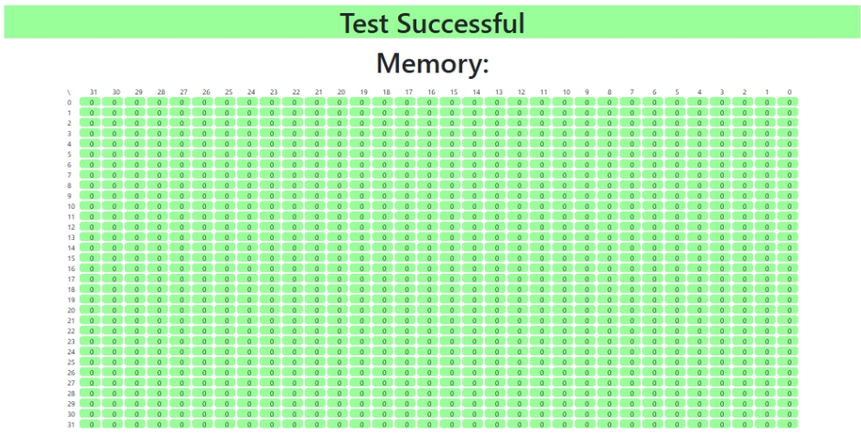

Figure 7.4: GUI of the verification platform displaying automated error mapping and testbench status metrics.

The operational flow of the verification framework is divided into three key automated phases:

1. A custom Matlab engine generates the specific test vectors required to stress the memory array, producing both the hardware-level stimulus files and a corresponding reference file containing the expected output states.

2. The testbench drives the stimulation sequence into the SRAM interface, recording the empirical read/write responses into a raw hardware output log.

3. A Python-based post-processing script cross-examines the hardware output log against the Matlab reference file. Any observed bit mismatches are isolated and flagged. The sorted data is then rendered via a dedicated GUI, as depicted in Figure 7.4, providing real-time visual error mapping.

The immediate next phase of this research involves validating the physical silicon prototype in the laboratory using this automated testing pipeline. Following initial benchtop calibration, the final milestone will consist of ion beam characterization to measure the single-event cross-section of both macros, proving the efficacy of this area-neutral radiation-hardening technique in aerospace environments.

7.2 Compiler-Driven SRAM Integration and Digital Flow Implementation

This subsection details the physical integration of three distinct Static Random-Access Memory (SRAM) types generated via the *SRAM-Compiler* software utility provided by the TSMC foundry. It must be noted in advance that internal architectural information, sub-circuit topologies, and detailed schematic images

of these SRAM macros and their components are deliberately omitted from this document, due to the Non-Disclosure Agreement (NDA) executed with TSMC for the utilization of the memory bit-cell libraries and compilation software.

The integrated architecture operates at a nominal clock frequency of 100 MHz and incorporates three different types of SRAM that communicate with the external test environment through a shared serial interface. Each of the three compiled blocks is configured with 256 rows by 32 bits, resulting in an individual storage capacity of 1 kByte for each implemented memory array.

The three types of SRAM examined in this study are:

- **HVT (High-Threshold-Voltage):** This variant is characterized by a higher transistor activation threshold voltage, which renders the circuits less affected by sub-threshold static leakage current contributions. However, this configuration increases overall dynamic consumption and slows down the timing response of the circuit. The higher activation threshold makes this topology natively more suitable for deployment in aerospace operating environments.

- **LVT (Low-Threshold-Voltage):** This variant features a lower transistor activation threshold, which yields faster timing loops and decreases active switching consumption, but makes the memory array significantly more vulnerable to static leakage current overheads.

- **UHD (Ultra-High-Density):** These SRAM blocks are engineered with extremely dense layout pitches, making them ideal for integrating large storage capacities within heavily constrained silicon areas.

The primary objective of this project phase is to verify under ion beam testing the native radiation resistance of the SRAM macros generated via the *SRAM-Compiler*. If these compiled structures prove to be sufficiently resilient against radiation-induced effects in aerospace environments, they could be deployed directly in future designs without requiring manual custom layout generation (*Analog-on-Top*), such as the approach presented in the previous subsection (Chapter 7.1. This methodology would enable the integration of significantly larger memory banks (in the order of kBytes) with smaller silicon area dimensions and superior integration with the standard digital logic components.

Indeed, the integration of an SRAM generated through the *SRAM-Compiler* is far more suitable for co-integration alongside a Microprocessor compared to a custom *Analog-on-Top* SRAM macro. This approach permits a suite of highly accurate full-chip timing and power simulations, as compiler-driven SRAMs are characterized by a complete set of Electronic Design Automation (EDA) views and front-end models, just like standard digital logic cells.

The *SRAM-Compiler* dynamically instantiates the physical macros based on user-defined parameters such as memory capacity, structural area dimensions

and the specific technological cell variant to be generated. Mirroring the standard digital design flow, a dedicated set of libraries contains the foundational digital primitives used to compose the compiled SRAM structure. However, SRAM macros feature highly stringent shape, bounding, and aspect-ratio constraints due to the rigid internal architecture required to operate a memory array. This highly bounded structural template is precisely what allows the *SRAM-Compiler* to dynamically generate diverse memory arrays.

The SRAM primitives included in these foundry libraries describe the physical behavioral modeling of entire macro-scale building blocks, such as memory matrices, sense amplifiers, line drivers, and multiplexers; this is because the cells within these specific libraries are engineered solely to compose SRAM structures. This contrasts sharply with standard digital cell libraries, which describe elementary logic gates used universally to implement any digital circuit described via Hardware Description Language (HDL). In other words, when implementing a specific highly structured module like a memory macro, the design tools do not require the wide expressive capability of gate-level primitives down to the single elementary logic gate.

For this reason, the libraries characterizing compiled SRAM cells are documented and structured differently than standard digital cells. This macro-block-level description is more suitable for modeling internal memory components and allows the design to undergo full-chip co-simulations alongside the rest of the digital logic.

However, the different data formats used to characterize standard digital cells versus macro compiled SRAM blocks require systematic modifications to the digital implementation flow to achieve a unified definition of the parameters utilized across the EDA tools.

Specifically, the *Floorplan* phase during the *Place&Route* routine demands a dedicated study and modification of the top-level *Power-Grid*, which depends heavily on the architecture of the specific SRAM array being integrated. This modification is required because SRAM macros implement a highly unique and proprietary internal power distribution structure. Furthermore, the foundry does not provide open documentation detailing this internal power strip routing architecture. Consequently, it was necessary to perform a careful layout study of the internal primitives within the digital SRAM libraries to understand their internal architecture. Due to the strict NDA constraints discussed above, further layout images and detailed descriptions of the internal SRAM architecture are omitted from this work.

Figure 7.5 illustrates the finalized post-routing physical layout obtained via the digital design flow on the left, cross-mapped with its physical allocation on the manufactured silicon die on the right.

Because an SRAM array exhibits a highly specialized operational behavior, it is necessary to design a dedicated *Vector-Based* simulation testbench. Indeed,

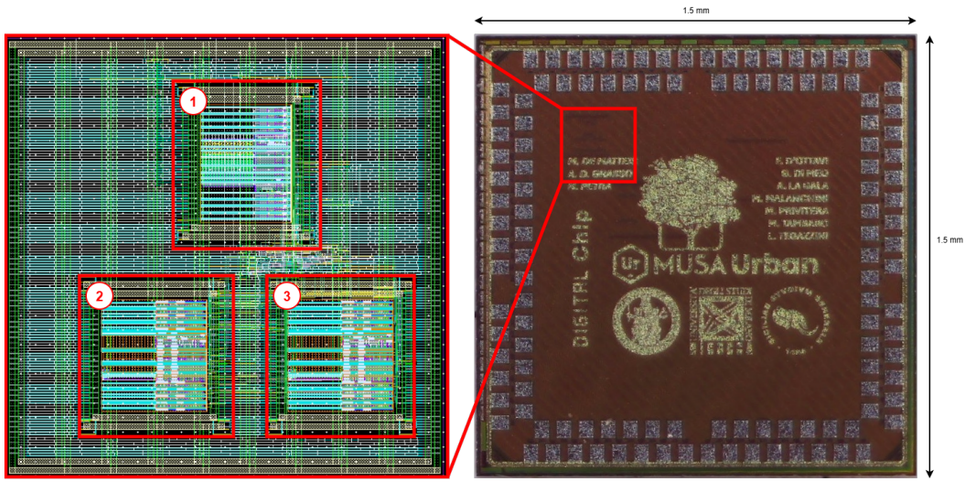

Figure 7.5: Post-*Place&Route* layout view (Left) and corresponding physical silicon die allocation (Right) of the triple compiled-SRAM subsystem. 1) HVT SRAM, 2) LVT SRAM 3) UHD SRAM.

standard statistical *Vector-Less* power estimation methods cannot be applied to these structures. *Vector-Less* simulations rely on statistical *Toggle-Rate* assumptions; setting generic statistical parameters severely overestimates the internal *Switching-Activity* of the SRAM, generating invalid power consumption figures. This error occurs because, in a worst-case scenario, an active SRAM cycle modifies at most a single memory row, which corresponds to just 32 bits.

To correctly set statistical parameters based on the storage capacity of the integrated memories, one would have to force a global *Toggle-Rate* down to a factor of 0.0039 (0.39 %) to reflect the true *Switching-Activity* of the SRAM array. However, enforcing such a low coefficient globally would completely underestimate the *Switching-Activity* of the rest of the digital peripheral circuit, failing to verify its power profile accurately. Therefore, it is mandatory to deploy *Vector-Based* power simulations, which extract the exact *Switching-Activity* from Value Change Dump (*Signoff*) files generated during active testbench execution.

The immediate next step of this project involves the execution of a testbench following the specialized measurement procedure developed for SRAM verification detailed in the previous section (Chapter 7.1. The testbench must be explicitly designed to allow robust characterization and power profiling across both virtual pre-silicon simulations and physical laboratory measurements using the experimental setup.

Finally, this testbench architecture is engineered to support the concluding milestone of the project, which consists of executing a complete ion beam characterization of the three compiled SRAMs. This test campaign will verify the native radiation resistance of compiler-driven foundry memories against the single-event effects caused by radiation exposure in aerospace flight environments.

7.3 Floating-Point Unit (FPU) Architecture Integration

This subsection presents the physical integration framework and back-end synthesis of a high-performance 500 MHz Floating-Point Unit (FPU). This computational block implements the standard M (integer multiplication and division) and F (single-precision floating-point arithmetic) extensions defined by the RISC-V ISA. The primary objective of this subsystem integration is to significantly augment the Microprocessor throughput of the baseline core developed in this thesis. By natively accelerating single-precision floating-point operations alongside hardware-accelerated multiplication, division, and square root instructions, this extension broadens the potential application space of the Microprocessor in high-performance embedded systems [98].

Architecturally, an FPU operating at a target frequency of 500 MHz represents a highly complex subsystem that introduces severe physical design and timing closure bottlenecks within the digital implementation flow.

Table 7.1: Gate Equivalent (GE) count for each constituent macro-block of the Floating-Point Unit (FPU).

Component	**Kilo Gate Equivalent [kGE]**
Multiplier	27.4
Divider	9.3
SQRT	4.7
Rounding	0.1
Routing	13.5
Total	55

Table 7.1 provides a detailed structural breakdown of the gate complexity expressed in kilo-Gate Equivalents (kGE) and physical area metrics collected after the initial physical *Synthesis* and *Place&Route* compilation. Evaluating the post-routing cell distribution demonstrates the structural density of the design, which demands a highly strategized physical layout *Floorplan* to ensure routability and prevent signal congestion.

At an operating frequency of 500 MHz, the architectural complexity significantly compounds physical convergence challenges. Throughout the progressive stages of the *Place&Route* flow, the physical layout undergoes major topological transformations driven by the EDA optimization engines, as illustrated in the side-by-side progression of Figure 7.6.

During these successive optimization iterations, a massive volume of structural elements such as clock distribution buffers, skew-balancing cells, and complex multi-layer routing tracks are dynamically inserted into the core canvas. This structural evolution is clearly visible when comparing the initial cell placement boundary (Figure 7.6, left) to the mid-flow CTS buffer configuration

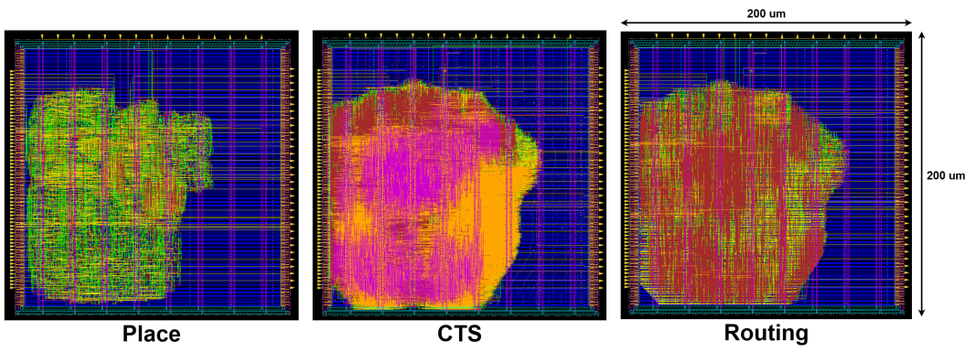

Figure 7.6: Physical transformation of the FPU layout macro across successive *Place&Route* optimization milestones: (Left) Post-*Place*; (Center) Post-CTS; (Right) Post-*Routing*.

(Figure 7.6, center) and the final post-Routing interconnect matrix (Figure 7.6, right).

To accommodate this high level of automated net insertion and ensure timing convergence, the initial *Floorplan* must deliberately maintain a conservative cell utilization density, leaving a significant percentage of free placement area unallocated. Because the 500 MHz clock period leaves narrow timing margins, multiple logic paths naturally emerge as highly critical. To fix *Setup* and *Hold* violations, the optimization engine requires ample unconstrained physical space to insert high-drive strength sizing variants, repeaters, and detour routing paths without inducing local routing blockages.

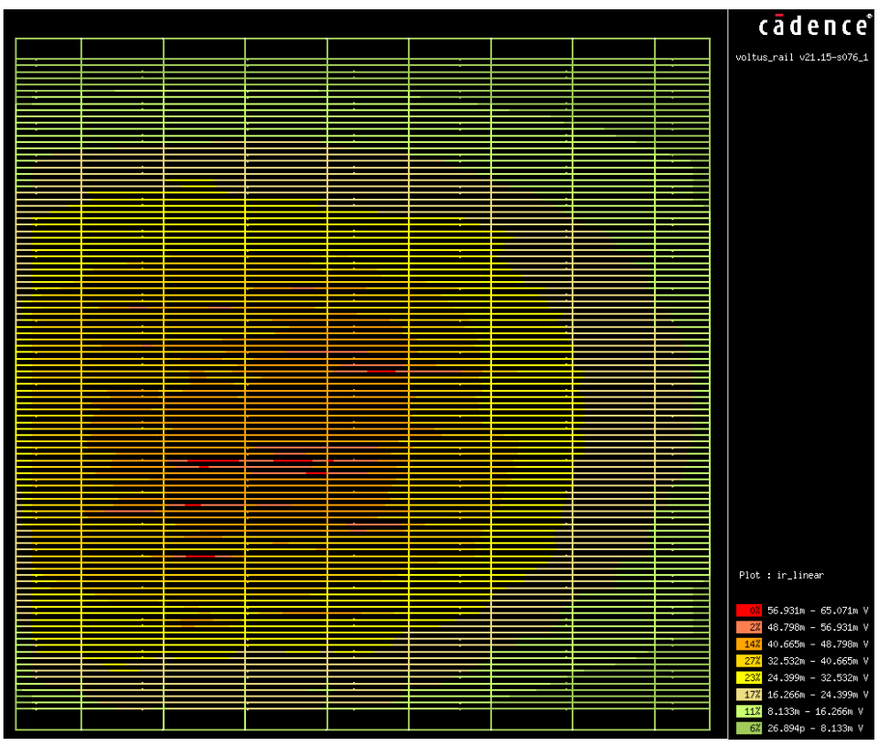

Figure 7.7: *IR-Drop* voltage degradation map across the consolidated FPU *Power-Grid*.

These physical constraints elevate the floorplanning phase and the *Power-*

Grid co-design into highly critical tasks. The top-level *Power-Grid* must be synthesized with exceptional structural robustness and routing density to suppress dynamic voltage droops and prevent excessive *IR-Drop* degradation under peak switching current conditions.

Table 7.2: Architectural parameters and performance of the proposed Floating-Point Unit (FPU).

Parameters	**Value**	**Units**
Technology Node	28nm CMOS Bulk HPC+ TSMC	**[-]**
Clock Frequency	500	**[MHz]**
Size	200 x 200	**[μm]**
Area	0.04	**[mm^2]**
Stripe Width	1	**[μm]**
Stripe Dinstance	22	**[μm]**
Stripe Number	8	**[μm]**
Ring Width	2	**[μm]**
Corner	TT, 0.9 V, 25 °C	**[-]**
Static Current	0.031	**[mA]**
Dynamic Current	5.322	**[mA]**
Total Current	5.353	**[mA]**
IR-Drop	57	**[mV]**

To achieve this reliability target, a multi-tier supply ring infrastructure was implemented within the upper metallization layers. Outer power rings with a width of 2 μm were routed using Metal 7 (M7) and Metal 8 (M8) layers to surround the FPU macro block. This boundary infrastructure is paired with 8 vertical internal supply straps fabricated in Metal 7 with a width of 1 μm, distributed uniformly across the core canvas at an iterative pitch of 22 μm (Table 7.2).

The performance of this customized *Power-Grid* topology is validated by the power integrity analysis illustrated in Figure 7.7. The simulation demonstrates a peak localized *IR-Drop* of only 57 mV, comfortably satisfying the strict *Signoff* limit of less than 100 mV.

Figure 7.8 shows the finalized post-routing GDSII layout of the integrated FPU macro generated via the digital flow. Actively executing the full physical design flow for this block is fundamental to expose deeply embedded critical paths and introduce targeted architectural or layout-level timing corrections.

Given the massive physical alterations that the netlist undergoes during *Place&Route* execution, complete back-end integration remains the only valid methodology to accurately simulate, analyze, and *Signoff* the core. This physical flow is mandatory to guarantee functional logic correctness, satisfy timing margin constraints, and validate electrical grid integrity under maximum current loads.

The implementation presented here represents the successful first-pass physical integration and verification phase of the independent FPU subsystem macro. Having confirmed the stability of the digital implementation flow

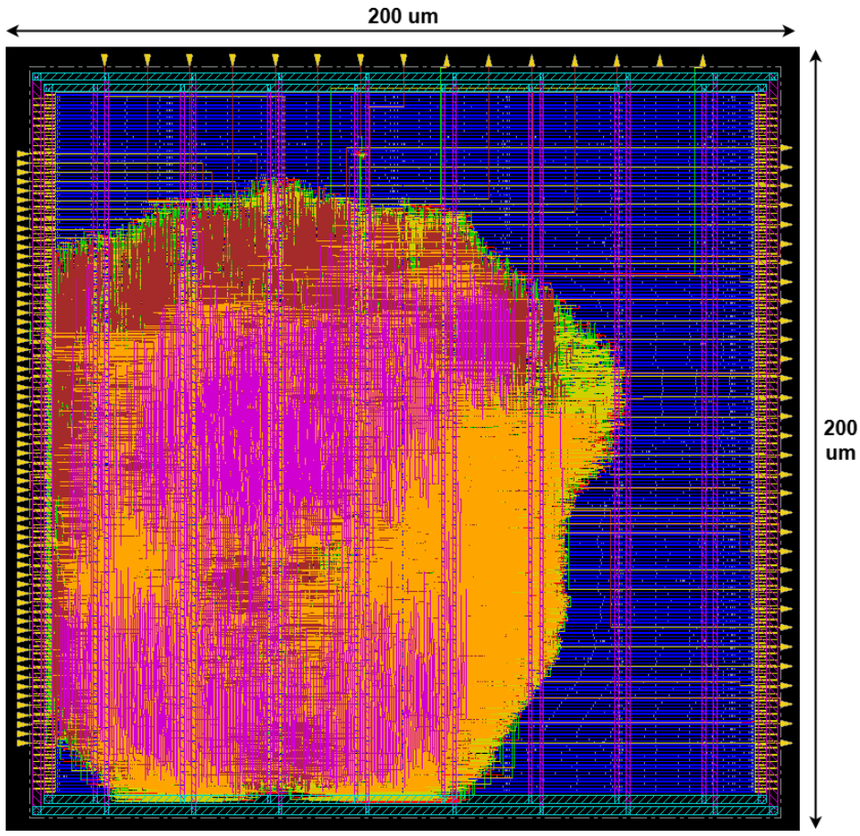

Figure 7.8: Finalized GDSII layout macro of the integrated 500 MHz RISC-V FPU subsystem.

and validated the baseline Microprocessor core architecture throughout this thesis, the next step of this research pipeline will focus on the full chip-level co-integration of this FPU directly into the central core datapath.

This subsequent development phase will aim toward full Application-Specific Integrated Circuits (ASIC) silicon manufacturing, enabling empirical laboratory verification, power-performance profiling and characterization utilizing the identical automated benchtop instrumentation setup developed in this work (Chapter 5).

7.4 Digital Logic Integration and Verification of a 640 MHz Aerospace PLL

This subsection presents the development, physical integration and empirical characterization of the digital logic control infrastructure for a high-frequency 640 MHz Phase-Locked Loop (PLL). Fabricated in a 28 nm Bulk CMOS HPC+ process node, this mixed-signal macro block is explicitly engineered to fulfill the stringent clock-stability and reliability mandates of aerospace operating environments.

Figure 7.9 illustrates the comprehensive architectural block diagram of the PLL, delineating the exact functional boundary of the synthesized digital

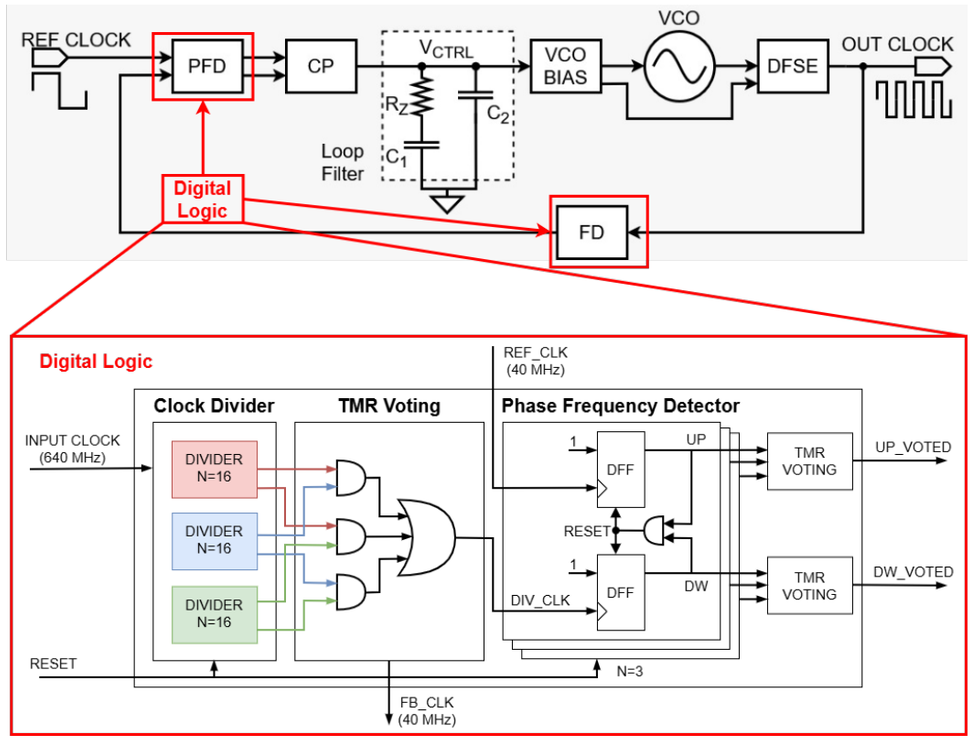

Figure 7.9: Architectural block diagram of the mixed-signal PLL highlighting the structural sub-modules of the synthesized digital logic.

controller. Within this *Analog-on-Top* macro framework, the digital logic is tasked with capturing the high-frequency output generated clock (640 MHz), establishing the synchronized feedback loop, and performing phase-frequency evaluation against an external *Reference Clock* (40 MHz).

Operationally, the digital subsystem utilizes a programmable multi-stage frequency divider to scale down the analog clock output, generating a localized *Feedback Clock* (40 MHz) phase-aligned and frequency-matched to the input *Reference Clock*. A high-speed Phase-Frequency Detector (PFD) then cross-examines the phase error between *Reference Clock* and *Feedback Clock*, dynamically driving digital control primitives (UP and $DOWN$ pulses) to regulate the upstream analog charge-pump and loop-filter networks.

To guarantee high-reliability operations against radiation-induced SET and SEU, a rigorous Triple Modular Redundancy (TMR) structural mitigation strategy was developed for both the frequency divider and the PFD. Rather than applying a single global voting block, a distributed dual-node majority-voting scheme was executed (Figure 7.9).

Majority voters are inserted both at the output boundary of the sequential dividing register chains and immediately downstream of the combinational PFD logic paths. This voting stage isolates the critical, high-speed combinational nodes, arresting soft-error propagation before it can disrupt the analog biasing loops and cause phase-lock dissociation.

Figure 7.10 provides a detailed view of the gate-level netlist obtained at

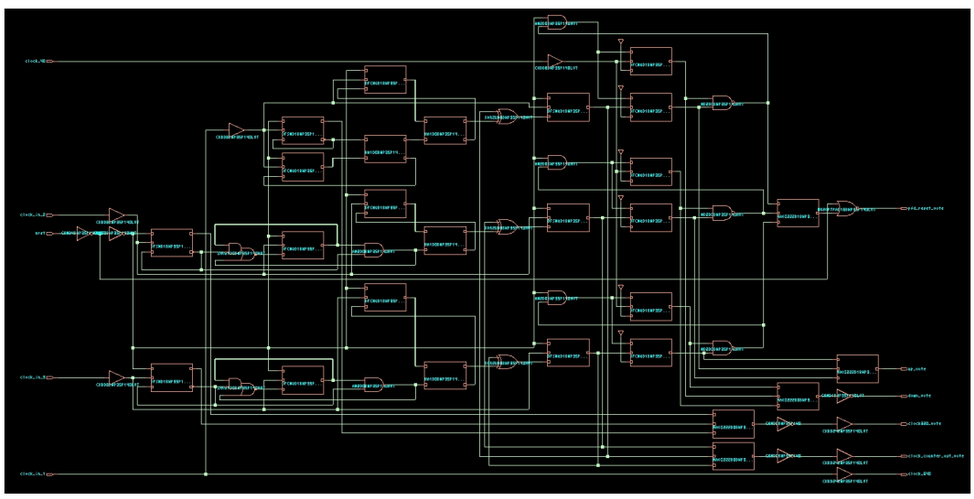

Figure 7.10: Detailed post-*Synthesis* (Cadence Genus) gate-level schematic of the PLL digital logic.

the conclusion of the digital *Synthesis* phase. During *Synthesis* execution, maintaining structural redundancy represents a major EDA flow bottleneck. Standard commercial optimization algorithms are natively programmed to identify parallel redundant paths as dead logic, systematically pruning out TMR structures and majority voters to minimize area.

To prevent this optimization behavior, specialized hierarchical constraints (*dont_touch*) were hard-coded into the *Synthesis* scripts, forcing the EDA engine to preserve the exact triplicated topology. This constraint framework required a structural, low-level hardware description language (HDL) modeling methodology that relies on structural gate-level instantiations rather than abstract behavioral descriptions.

Operating at a target frequency of 640 MHz demands meticulous timing constraint definitions during the logical *Synthesis* and physical *Place&Route* steps. Because the control paths propagate either high-speed clock signals or critical asynchronous lines with tighter timing margins than the clock itself such as the PFD UP and $DOWN$ error signals standard clock definitions are insufficient. This required complex Multi-Cycle path constraints, explicit Clock Tree Synthesis (CTS) routing rules, and dedicated generated-clock properties to guarantee precise timing closure across cross-dependent clock domains.

To protect the sequential storage elements from SEU phenomena, where a single ionizing particle strike flips adjacent memory nodes, a strict physical cell spacing constraint of at least 5 μm was enforced during the placement phase. Fortuitously, the insertion of the dual distributed majority-voting networks naturally decouples the sequential divider blocks from the PFD. This topological separation allows the *Place&Route* engine to tightly packet the redundant standard cells while satisfying the radiation spacing rules, minimizing the physical area overhead typically induced by structural cell placement separation.

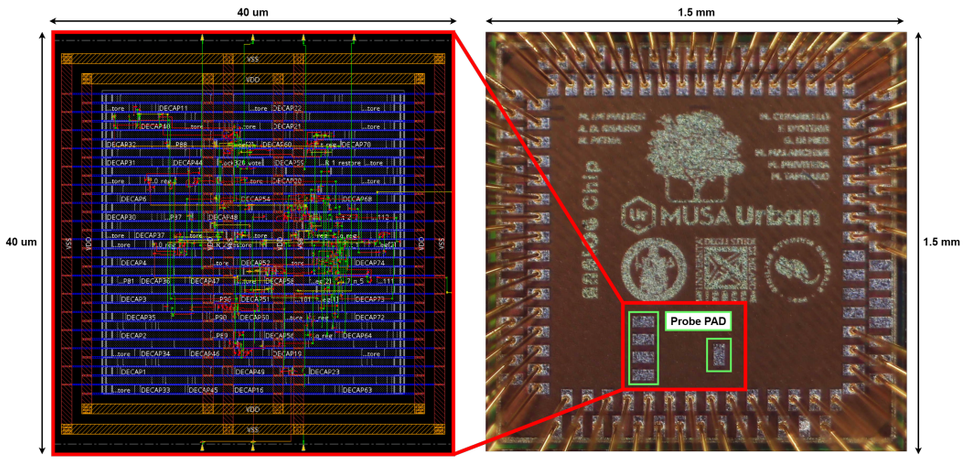

Figure 7.11: Post-*Routing* physical layout (left) and corresponding silicon die placement (right) of the 28 nm digital PLL logic, with green boxes highlighting the internal *Probe-PAD* positions for high-frequency testing.

The finalized post-*Routing* physical layout of the digital controller macro is illustrated in Figure 7.11 (left), alongside its physical co-integration on the fabricated 28 nm Bulk CMOS HPC+ silicon die (right), highlighting the *Probe-PAD* dedicated to post-silicon testing.

Table 7.3: Architectural parameters and performance of the proposed Phase-Locked Loop (PLL).

Parameters	**Value**	**Units**
Technology Node	28nm CMOS Bulk HPC+ TSMC	[-]
Clock Frequency	640	[**MHz**]
Size	40 x 40	[μ**m**]
Area	0.0016	[**mm**^{2}]
Gate Equivalent	625	[-]
Corner	TT, 0.9 V, 25 °C	[-]
Static Current	0.001	[**mA**]
Dynamic Current	0.388	[**mA**]
Total Current	0.389	[**mA**]
IR-Drop	22	[**mV**]

Table 7.3 presents the primary physical, structural, and electrical parameters extracted from post-layout back-end *Signoff* simulations.

Because the complete loop constitutes an *Analog-on-Top* mixed-signal design, multi-domain validation must be carried out within a high-precision analog verification environment. To capture the complex timing interplay of the digital logic within the continuous-time analog domain, a parasitic-aware mixed-signal simulation flow was implemented.

During the digital *Signoff* phase, Standard Parasitic Extraction (*.SPEF*) and Standard Delay Format (*.SDF*) libraries were generated with a customized delay resolution scaled to the femtosecond (fS) level. These extracted parasitic and interconnect latency models were then imported directly into the

Cadence Spectre simulator, enabling high-fidelity spice-level co-simulations of the complete mixed-signal loop.

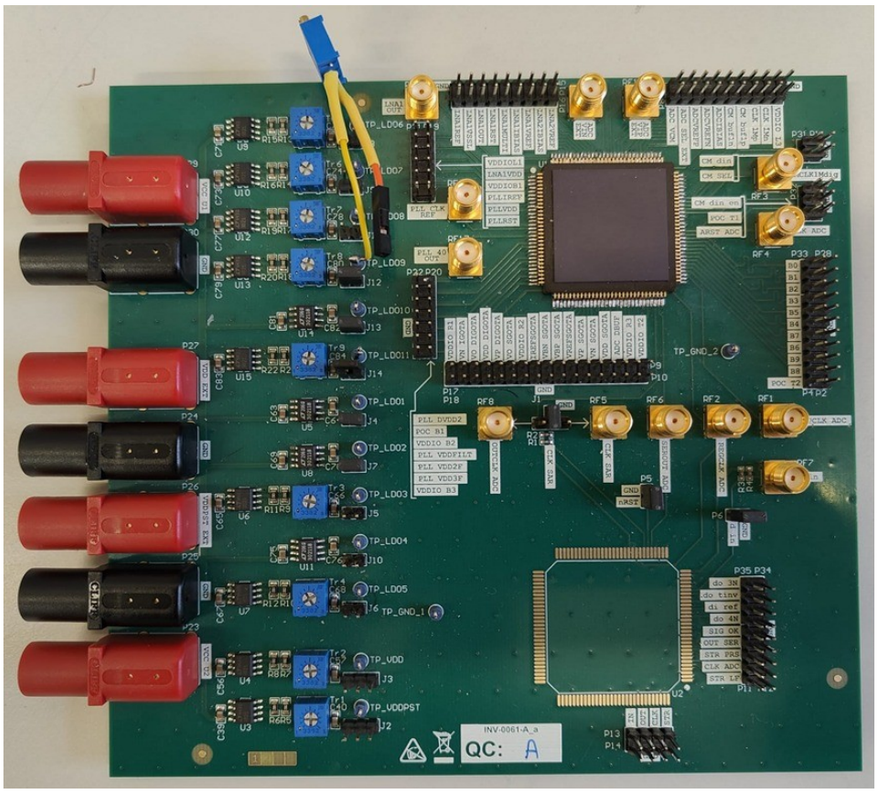

Figure 7.12: Custom-designed characterization PCB for benchtop laboratory measurement.

To conduct empirical verification of the manufactured prototype, a high-frequency characterization Printed Circuit Board (PCB) was developed, as shown in Figure 7.12. Directly measuring a standalone 640 MHz clock wave via conventional benchtop instruments is prevented by hardware filtering constraints; standard packaging options and the chip-level *Padring* infrastructure introduce severe low-pass parasitic characteristics that distort digital signals above 250 MHz. Furthermore, reducing the input *Reference Clock* frequency to scale down the output is not feasible, as operating outside the designed tuning window prevents the Voltage-Controlled Oscillator (VCO) from achieving stable phase-lock.

To validate circuit operation within these constraints, an alternative testing methodology was developed by routing the internal *Feedback Clock* to an external test pad. The *Feedback Clock* propagates through the entire internal logic and dividing path at a highly manageable frequency of 40 MHz. Measuring optimal signal performance; specifically phase jitter, duty-cycle stability, and balanced rise/fall times, on the *Feedback Clock* line serves as a first-order validation of successful phase-locking and digital controller tracking.

Figure 7.13 provides a direct time-domain comparison between the simulated *Feedback Clock* waveform and the empirical signal captured via laboratory instrumentation. The two profiles exhibit a high degree of correlation. The

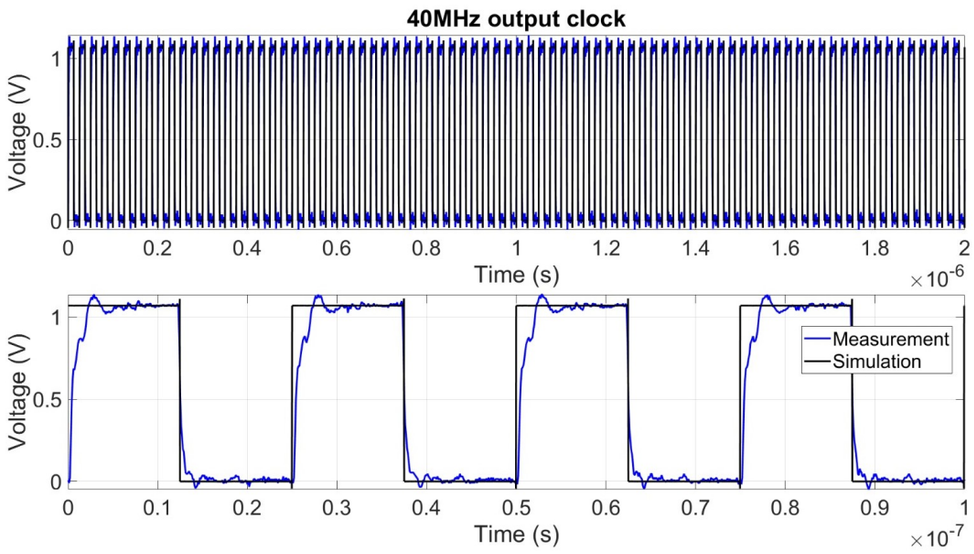

Figure 7.13: Time-domain waveform comparison between the simulated *Feedback Clock* and the empirical benchtop laboratory measurement.

slight distortion observed on the rising edge of the measured physical signal is traced back to trace-level parasitic impedances native to the test PCB rather than silicon-level macro degradation.

Directly extracting the raw 640 MHz output clock without package attenuation requires a dedicated micro-probing setup. As indicated in Figure 7.11, specialized, high-bandwidth internal *Probe-PAD* were embedded directly inside the core layout to support direct RF needle contact via a *Probe-Station*. This instrumentation setup operates directly on the *Naked-Die* (de-capped package), providing the Gigahertz-level analog bandwidth necessary to calculate phase jitter, duty-cycle distortion, and rise/fall transitions.

To execute these micro-probing routines, an ultra-flat characterization PCB was designed to interface with the *Probe-Station* chuck. The bottom layer of this board is engineered to be perfectly planar, ensuring a rigid vacuum seal with the suction pump of the station. This configuration guarantees zero physical movement of the PCB during the micrometer-scale alignment of the high-frequency probes.

The immediate next steps to complete this research program will focus on executing the direct 640 MHz *Probe-PAD* characterization via the vacuum-sealed probe setup, followed by a ion irradiation campaign to empirically demonstrate the single-event cross-section mitigation achieved by the dual-node TMR digital block.

Chapter 8

Conclusions and Future Research Lines

This thesis presents the comprehensive design procedure, physical implementation, and experimental characterization, via post-layout simulations and laboratory electrical testing, of a 28 nm Bulk Complementary Metal-Oxide-Semiconductor (CMOS) Microprocessor based on the Reduced Instruction Set Computer - Five (RISC-V) architecture for aerospace applications.

The developed Microprocessor design incorporating the RISC-V RV32I (Base Integer Instruction Set Architecture) operating at a nominal clock frequency of 100 MHz, occupies a physical silicon footprint of 0.05 mm^2 (with dimensions of 0.225 mm $\times$ 0.225 mm). This footprint is equivalent to 52,682.25 Gate Equivalents (GE) globally, and 13,649 GE when isolating the core (*RV32I-Core*). At the physical layer, the macro comprises 33,221 digital standard cells, translating to approximately 210,729 integrated CMOS transistors.

To satisfy the extreme reliability constraints of the aerospace radiation environment, the Microprocessor was explicitly engineered to mitigate radiation-induced degradation. Specifically, the solutions adopted to mitigate the impact of radiation include:

- **Solutions at the technology level:** The selection of a 28 nm Bulk CMOS process node over an Fully Depleted Silicon On Insulator (FD-SOI) alternative, which was explicitly driven by targeted optimizations to mitigate Total Ionizing Dose (TID) effects.

- **Solutions at the logic level:** The implementation of a specialized digital flow was driven by dedicated power optimizations. This effectively mitigates the leakage current inherent to the Bulk technology while increasing the intrinsic radiation hardness of the logic gates.

- **Solutions at the layout level:** The enforcement of a dense, robust *Power-Grid* alongside a rigorous Multi-Corner Multi-Mode (MCMM) timing closure flow to handle dynamic voltage fluctuations (minimizing

the *IR-Drop*), effectively countering transient supply instabilities and failures caused by Single-Event Effects (SEE) at the back-end level.

Given its reprogrammable nature, the verification and characterization of the circuit require a specific standardized testbench and validation strategy. This setup must ensure functional correctness without introducing statistical bias and validate reliability despite the impossibility of performing exhaustive verification of all instruction sequences.

To abstract the underlying hardware complexity, a unified verification framework was implemented by combining *Vector-Less* analysis, *Vector-Based* execution, and a hardware *Debug-Module* directly within the embedded testbench firmware. This structured environment enables identical software verification routines to be executed seamlessly across both virtual pre-silicon engineering phases (such as back-end *Signoff* and *Power-Analysis*) and post-silicon physical laboratory testing. Consequently, this unified testing framework provides a direct cross-correlation between simulated architectural performance models and empirical physical measurements.

The baseline *Vector-Less Power-Analysis* evaluated the circuit under worst-case switching conditions by forcing a global *Toggle-Rate* of 50 % alongside an *Increase Switching Consumption* factor of 30 %. Under these severe constraints, the core exhibited a static leakage current of 0.012 mA (representing a highly suppressed 0.44 % static-to-dynamic power contribution ratio), a dynamic current consumption of 2.7 mA, and a peak localized *IR-Drop* of only 39 mV. These metrics successfully validate the effectiveness of the physical layout techniques embedded during the *Synthesis* and *Place&Route* phases. These techniques were engineered to mitigate SEE, suppress induced voltage ripples, and contain the inherently higher leakage current typical of highly scaled 28 nm bulk CMOS nodes compared to FD-SOI alternatives.

Furthermore, the synthesis of dedicated characterization firmware enabled the precise isolation and measurement of the average current consumption associated with every individual instruction within the RV32I Instruction Set Architecture (ISA). *Vector-Based* power simulations driven by this firmware yielded a static leakage current of 0.011 mA (empirically measured at 0.01 mA in the laboratory) and an average dynamic execution current of 1.776 mA per instruction (measured at 1.81 mA in the laboratory). This performance translates to a highly competitive energy efficiency figure of 15.988 pJ/Instruction in simulation, closely mirroring the 16.287 pJ/Instruction demonstrated during physical benchtop characterization.

The experimental results obtained from both simulations and laboratory electrical measurements validate the successful achievement of this thesis's objectives. Comparative analysis demonstrates that the proposed Microprocessor implementation is highly competitive, outperforming the average of state-of-the-art of the comparable Microprocessors reported in recent literature.

Specifically, it delivers a superior energy-area trade-off, thereby validating the effectiveness of the proposed design integration methodology for high-reliability applications.

The work presented in this dissertation establishes the first foundational step required to realize a fully qualified, mission-ready aerospace RISC-V Microprocessor, validating both the custom architecture and the digital flow.

To transition this core into a flight-certified system, the immediate next milestone requires completing the design, physical integration, and verification of the auxiliary computational and mixed-signal subsystems introduced at the conclusion of this thesis (Chapter 7). Integrating the custom Floating Point Unit (FPU) will expand the Microprocessor's mathematical throughput; deploying the compiler-driven and custom Rad-Hard-by-Design (RHBD) Static Random-Access Memory (SRAM) blocks will optimize on-chip memory storage; and the digital Phase-Locked Loop (PLL) will distribute a highly stable clock reference.

Following the standalone calibration of these subsystems, consecutive radiation test campaigns under ion beams will be mandatory to measure the single-event cross-sections and empirical radiation tolerance of each individual module. This research program will culminate in the monolithic integration of all sub-systems into a unified System-on-Chip (SoC), followed by a final, full-scale irradiation campaign to achieve radiation hardness qualification for aerospace missions.

References

[1] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA," RISC-V International, Doc. Version 20191213, Dec. 2019.

[2] B. A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture," RISC-V International, Doc. Version 20211203, Dec. 2021.

[3] D. Seo, et al., "Total Ionizing Dose Effect on Ring Oscillator Frequency in 28-nm FD-SOI Technology," in IEEE Electron Device Letters, vol. 39, no. 11, pp. 1728-1731, Nov. 2018, doi: 10.1109/LED.2018.2872345.

[4] Yan, et al., "Simulation of Total Ionizing Dose (TID) Effects Mitigation Technique for 22 nm Fully-Depleted Silicon-on-Insulator (FDSOI) Transistor." IEEE Access 8 (2020): 154898-154905. https://doi.org/10.1109/access.2020.3018714.

[5] Li, Zong-Qiang, et al. "Comparison of Total Ionizing Dose Effects in 22-nm and 28-nm FD SOI Technologies." Electronics, 2022. https://doi.org/10.3390/electronics11111757.

[6] Acuña, A., et al. "A Mixed Method to Mitigate the TID Effects on 28nm FDSOI Transistors." 2020 20th European Conference on Radiation and Its Effects on Components and Systems (RADECS), 2020, pp. 1-6. https://doi.org/10.1109/radecs50773.2020.9857702.

[7] Bonaldo, S., et al. "Ionizing-Radiation Response and Low-Frequency Noise of 28-nm MOSFETs at Ultrahigh Doses." IEEE Transactions on Nuclear Science, vol. 67, 2020, pp. 1302-1311. https://doi.org/10.1109/tns.2020.2981881.

[8] Traversi, S. M. I. G., et al. "Ionizing Radiation Effects of 3-Grad TID on Analog and Noise Performance of 28-nm CMOS Technology." IEEE Transactions on Nuclear Science, vol. 72, 2025, pp. 3343-3350. https://doi.org/10.1109/tns.2025.3542230.

[9] G. Quilligan et al., "TID and Heavy-Ion Performance of an RHBD Multichannel Digitizer in 180-nm CMOS," in IEEE Transactions on Nuclear Science, vol. 68, no. 7, pp. 1414-1422, July 2021, doi: 10.1109/TNS.2021.3080179.

[10] Zhu, M.,et al., Radiation-hardened and repairable integrated circuits based on carbon nanotube transistors with ion gel gates. Nat Electron 3, 622–629 (2020). https://doi.org/10.1038/s41928-020-0465-1.

[11] P. I. Vaz, et al., "Improving TID Radiation Robustness of a CMOS OxRAM-Based Neuron Circuit by Using Enclosed Layout Transistors," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 29, no. 6, pp. 1122-1131, June 2021, doi: 10.1109/TVLSI.2021.3067446.

[12] Colpi, et al., "LISA Definition Study Report." (2024).

[13] Seoane, et al., "Astrophysics with the Laser Interferometer Space Antenna." Living Reviews in Relativity 26 (2022): 1-328. https://doi.org/10.1007/s41114-022-00041-y.

[14] Auclair, P., et al. "Cosmology with the Laser Interferometer Space Antenna." Living Reviews in Relativity, vol. 26, 2022, pp. 1-254. https://doi.org/10.1007/s41114-023-00045-2.

[15] Inchauspé, H., et al. "Measuring gravitational wave memory with LISA." Physical Review D, 2024. https://doi.org/10.1103/physrevd.111.044044.

[16] Khoo, L., et al. "Multispacecraft Observations of a Widespread Solar Energetic Particle Event on 2022 February 15–16." The Astrophysical Journal 963 (2024). https://doi.org/10.3847/1538-4357/ad167f.

[17] Genova, A.,et al., "Geodesy, Geophysics and Fundamental Physics Investigations of the BepiColombo Mission." Space Science Reviews 217 (2021). https://doi.org/10.1007/s11214-021-00808-9.

[18] Benkhoff, J., et al. "BepiColombo - Mission Overview and Science Goals." Space Science Reviews, vol. 217, 2021. https://doi.org/10.1007/s11214-021-00861-4.

[19] Mangano, V., et al. "BepiColombo Science Investigations During Cruise and Flybys at the Earth, Venus and Mercury." Space Science Reviews, vol. 217, 2021, https://doi.org/10.1007/s11214-021-00797-9.

[20] Fletcher, et al., "Jupiter Science Enabled by ESA's Jupiter Icy Moons Explorer." Space Science Reviews 219 (2023). https://doi.org/10.1007/s11214-023-00996-6.

[21] Masters, et al., "Magnetosphere and Plasma Science with the Jupiter Icy Moons Explorer." Space Science Reviews 221 (2025). https://doi.org/10.1007/s11214-025-01148-8.

[22] Van Hoolst, T., et al. "Geophysical Characterization of the Interiors of Ganymede, Callisto and Europa by ESA's JUpiter ICy moons Explorer." Space Science Reviews, vol. 220, 2024. https://doi.org/10.1007/s11214-024-01085-y.

[23] Boutonnet, A., et al. "Designing the JUICE Trajectory." Space Science Reviews, vol. 220, 2024. https://doi.org/10.1007/s11214-024-01093-y.

[24] Bebesi, Zsofia. "Exploration of Jupiter's Galilean satellites and their complex interactions with the Jovian magnetosphere: Hungary's participation in ESA's Jupiter Icy Moons Explorer mission." The European Physical Journal Special Topics, vol. 234, 2025, pp. 3029 - 3036. https://doi.org/10.1140/epjs/s11734-025-01467-5.

[25] Pappalardo, R., et al. "Science Overview of the Europa Clipper Mission." Space Science Reviews, vol. 220, 2024. https://doi.org/10.1007/s11214-024-01070-5.

[26] Vance, S., et al. "Investigating Europa's Habitability with the Europa Clipper." Space Science Reviews, vol. 219, 2023. https://doi.org/10.1007/s11214-023-01025-2.

[27] Howell, S., and R. Pappalardo. "NASA's Europa Clipper—a mission to a potentially habitable ocean world." Nature Communications, vol. 11, 2020. https://doi.org/10.1038/s41467-020-15160-9.

[28] Roberts, J. H., et al. "Exploring the Interior of Europa with the Europa Clipper." Space Science Reviews, vol. 219, 2023. https://doi.org/10.1007/s11214-023-00990-y.

[29] Cangahuala, L., et al. "Europa Clipper Mission Design, Mission Plan, and Navigation." Space Science Reviews, vol. 221, 2025. https://doi.org/10.1007/s11214-025-01140-2.

[30] Becker, T., et al. "Exploring the Composition of Europa with the Upcoming Europa Clipper Mission." Space Science Reviews, vol. 220, 2024. https://doi.org/10.1007/s11214-024-01069-y.

[31] Matteini, L., et al. "Parker Solar Probe: Four Years of Discoveries at Solar Cycle Minimum." Space Science Reviews, vol. 219, 2023, pp. 1-140. https://doi.org/10.1007/s11214-023-00952-4.

[32] Sebastián, A., et al. "An orbital model for the Parker Solar Probe mission: classical vs relativistic effects." Advances in Space Research, 2022. https://doi.org/10.1016/j.asr.2022.05.037.

[33] Livi, R., et al. "The Solar Probe ANalyzer—Ions on the Parker Solar Probe." The Astrophysical Journal, vol. 938, 2021. https://doi.org/10.3847/1538-4357/ac93f5.

[34] Q. M. Khan, et al., "A Comparative Performance Analysis of 6T & 9T SRAM Integrated Circuits: SOI vs. Bulk," in IEEE Letters on Electromagnetic Compatibility Practice and Applications, vol. 4, no. 2, pp. 25-30, June 2022, doi: 10.1109/LEMCPA.2022.3163963.

[35] A. O.,et al., "OFF-State leakage current mechanisms in bulkSi and SOI MOSFETs and their impact on CMOS ULSIs standby current," in IEEE Transactions on Electron Devices, vol. 48, no. 9, pp. 2050-2057, Sept. 2001, doi: 10.1109/16.944195.

[36] Khan, Qazi Mashaal, et al., "A Comparative Study of On-Chip CMOS SH Voltage Sensors for Power Integrity: SOI vs. Bulk." 2021 IEEE International Joint EMC/SI/PI and EMC Europe Symposium, 2021, pp. 911-916. https://doi.org/10.1109/emc/si/pi/emceurope52599.2021.9559242.

[37] Waterman, Andrew. "Design of the RISC-V Instruction Set Architecture." (2016).

[38] A. Waterman, et al., "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213", RISC-V Foundation, December 2019.

[39] Kalapothas, Stavros, et al. "A Survey on RISC-V-Based Machine Learning Ecosystem." Inf., vol. 14, 2023, pp. 64. https://doi.org/10.3390/info14020064.

[40] RISC-V. (2026, April 8). Home - RISC-V International. RISC-V International. https://riscv.org/.

[41] Kannaujiya, Aryan, and A. Shah. "Radiation Effects in VLSI Circuits – Part I: Historical Perspective." IETE Technical Review, vol. 41, 2024, pp. 716 - 735. https://doi.org/10.1080/02564602.2024.2380904.

[42] Aguiar, Vitor A. P., et al. "Radiation-Induced Effects on Semiconductor Devices: A Brief Review on Single-Event Effects, Their Dynamics, and Reliability Impacts." Chips, 2025. https://doi.org/10.3390/chips4010012.

[43] Luza, Lucas Matana, et al. "Impact of Atmospheric and Space Radiation on Sensitive Electronic Devices." 2022 IEEE European Test Symposium (ETS), 2022, pp. 1-10. https://doi.org/10.1109/ets54262.2022.9810454.

[44] Mattos, André M. P., et al. "Investigation of Single-Event Effects for Space Applications: Instrumentation for In-Depth System Monitoring." Electronics, 2024. https://doi.org/10.3390/electronics13101822.

[45] Gao, Tianzhi, et al. "Simulation of Total Ionizing Dose Effects Technique for CMOS Inverter Circuit." Micromachines, vol. 14, 2023. https://doi.org/10.3390/mi14071438.

[46] Tselios, K., et al. "On the Distribution of Single Defect Threshold Voltage Shifts in SiON Transistors." IEEE Transactions on Device and Materials Reliability, vol. 21, 2021, pp. 199-206. https://doi.org/10.1109/tdmr.2021.3080983.

[47] Yao, Zihan, et al. "Material Synthesis and Device Aspects of Monolayer Tungsten Diselenide." Scientific Reports, vol. 8, 2018. https://doi.org/10.1038/s41598-018-23501-4.

[48] Kao, K., et al. "Numerical Simulations of Gate-Granularity- Induced Subthreshold Characteristics Deterioration of MOSFETs Magnified at Cryogenic Temperatures." IEEE Access, vol. 12, 2024, pp. 169748-169754. https://doi.org/10.1109/access.2024.3497933.

[49] Furuta, J. "Radiation-induced Soft Errors in Digital Circuits." 2024 International VLSI Symposium on Technology, Systems and Applications (VLSI TSA), 2024, pp. 1-2. https://doi.org/10.1109/vlsitsa60681.2024.10546387.

[50] Han, Jin-Woo, et al. "Single Event Hard Error due to Terrestrial Radiation." 2021 IEEE International Reliability Physics Symposium (IRPS), 2021, pp. 1-6. https://doi.org/10.1109/irps46558.2021.9405177.

[51] Gobatto, L., et al. "Early Radiation-Induced Soft-Error Assessment of Arm Cortex-M SoCs Through Fault Injection." IEEE Transactions on Device and Materials Reliability, vol. 25, 2025, pp. 45-53. https://doi.org/10.1109/tdmr.2024.3501193.

[52] Wu, Lei, et al. "Analysis of Displacement Damage Induced by Silicon-Ion Irradiation in SiC MOSFETs." IEEE Transactions on Nuclear Science, vol. 71, 2024, pp. 1370-1379. https://doi.org/10.1109/tns.2024.3408466.

[53] Alj, Antoine Salih, et al. "Total Ionizing Dose Effects on a CDTI-Based CCD-on-CMOS Through Buildup of Interface Traps and Oxide Charges." IEEE Transactions on Nuclear Science, vol. 71, 2024, pp. 1774-1782. https://doi.org/10.1109/tns.2024.3358381.

[54] Brewer, Rachel M., et al. "Total Ionizing Dose Responses of 22-nm FDSOI and 14-nm Bulk FinFET Charge-Trap Transistors." IEEE Transactions on Nuclear Science, vol. 68, 2021, pp. 677-686. https://doi.org/10.1109/tns.2021.3059594.

[55] Youssef, A. A., et al. "Compact Modeling of Single-Event Latchup of Integrated CMOS Circuit." IEEE Transactions on Nuclear Science, vol. 66, 2019, pp. 1510-1515. https://doi.org/10.1109/tns.2019.2920629.

[56] P. E. Dodd, et al., "Basic mechanisms and modeling of single-event upset in digital microelectronics," in IEEE Transactions on Nuclear Science, vol. 50, no. 3, pp. 583-602, June 2003, doi: 10.1109/TNS.2003.813129.

[57] D. Kobayashi, et al., "Scaling Trends of Digital Single-Event Effects: A Survey of SEU and SET Parameters and Comparison With Transistor Performance," in IEEE Transactions on Nuclear Science, vol. 68, no. 2, pp. 124-148, Feb. 2021, doi: 10.1109/TNS.2020.3044659.

[58] Aguiar, et al., "Mitigation and Predictive Assessment of SET Immunity of Digital Logic Circuits for Space Missions." Aerospace (2020). https://doi.org/10.3390/aerospace7020012.

[59] L. Gelmi, et al., "20 MeV·cm2/mg Linear Energy Transfer Radiation Tolerant Six-Transistor Static-Random-Access-Memory Cell in 28 nm CMOS Technology," 2025 20th International Conference on PhD Research in Microelectronics and Electronics (PRIME), Taormina, Italy, 2025, pp. 1-4, doi: 10.1109/PRIME66228.2025.11203670.

[60] A. H. Johnston, et al., "The effects of space radiation on linear integrated circuits," 2000 IEEE Aerospace Conference. Proceedings (Cat. No.00TH8484), Big Sky, MT, USA, 2000, pp. 363-369 vol.5, doi: 10.1109/AERO.2000.878509.

[61] Mauguet, M., et al. "Single-Event Latchup in a CMOS-Based ASIC Using Heavy Ions, Laser Pulses, and Coupled Simulation." IEEE Transactions on Nuclear Science, vol. 66, 2019, pp. 1516-1522. https://doi.org/10.1109/tns.2019.2920842.

[62] Yaqing, Chi, et al. "Characterization of Single-Event Upsets Induced by High-LET Heavy Ions in 16-nm Bulk FinFET SRAMs." IEEE Transactions on Nuclear Science, vol. 69, 2022, pp. 1176-1181. https://doi.org/10.1109/tns.2021.3127567.

[63] Caron, P., et al. "Physical Mechanisms of Proton-Induced Single-Event Upset in Integrated Memory Devices." IEEE Transactions on Nuclear Science, vol. 66, 2019, pp. 1404-1409. https://doi.org/10.1109/tns.2019.2902758.

[64] Bang, Minji, et al. "Mitigation of Single Event Upset Effects in Nanosheet FET 6T SRAM Cell." IEEE Access, vol. 12, 2024, pp. 130347-130355. https://doi.org/10.1109/access.2024.3457750.

[65] Aketi, Sai Aparna, et al. "SERAD: Soft Error Resilient Asynchronous Design Using a Bundled Data Protocol." IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 67, 2020, pp. 1667-1677. https://doi.org/10.1109/tcsi.2020.2965073.

[66] Frenkel, C., et al. "Comparative analysis of redundancy schemes for soft-error detection in low-cost space applications." 2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), 2016, pp. 1-6. https://doi.org/10.1109/vlsi-soc.2016.7753573.

[67] Kuentzer, F., et al. "Radiation Hardened Click Controllers for Soft Error Resilient Asynchronous Architectures." 2020 26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), 2020, pp. 78-85. https://doi.org/10.1109/async49171.2020.00020.

[68] Zheyu, Zhan, et al. "Single-Event Upset Monitor-Based Radiation-Hardened Latch for Multi-Node Upset." IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 72, 2025, pp. 1093-1097. https://doi.org/10.1109/tcsii.2025.3582493.

[69] Alacchi, Aurélien, et al. "Low Latency SEU Detection in FPGA CRAM With In-Memory ECC Checking." IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 70, 2023, pp. 2028-2036. https://doi.org/10.1109/tcsi.2023.3243644.

[70] Chen, Junchao, et al. "Solar Particle Event and Single Event Upset Prediction from SRAM-Based Monitor and Supervised Machine Learning." IEEE Transactions on Emerging Topics in Computing, vol. 10, 2022, pp. 564-580. https://doi.org/10.1109/tetc.2022.3147376.

[71] Schrape, O., et al. "Low Overhead Self-Correction in Radiation-Hardening-by-Design Triple Modular Redundancy Flip-Flops for Space Applications." 2025 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2025, pp. 1-6. https://doi.org/10.1109/dft66274.2025.11257570.

[72] Aketi, Sai Aparna, et al. "Single-Error Hardened and Multiple-Error Tolerant Guarded Dual Modular Redundancy Technique." 2018 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID), 2018, pp. 250-255. https://doi.org/10.1109/vlsid.2018.71.

[73] Banteywalu, S., et al. "A Novel Modular Radiation Hardening Approach Applied to a Synchronous Buck Converter." Electronics, 2019. https://doi.org/10.3390/electronics8050513.

[74] Miura, Y., et al. "Development of FF Circuits for Measures Against Power Supply Noise." 2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS), 2019, pp. 48-51. https://doi.org/10.1109/iolts.2019.8854447.

[75] Cheng, Silu, et al. "Effect of Static IR-Drop in Large-Scale Event-Based Vision Sensors." IEEE Sensors Journal, vol. 25, 2025, pp. 28323-28330. https://doi.org/10.1109/jsen.2025.3583394.

[76] Bhooshan, Rishi, et al. "Novel Power Grid Design Architecture to Improve IR Voltage Drop and EMI Shield." 2024 IEEE Joint International Symposium on Electromagnetic Compatibility, Signal Power Integrity: EMC Japan / Asia-Pacific International Symposium on Electromagnetic Compatibility (EMC Japan/APEMC Okinawa), 2024, pp. 556-559. https://doi.org/10.23919/emcjapan/apemcokinaw58965.2024.10585183.

[77] K. Asanović and D. A. Patterson, "Instruction Sets Should Be Free: The Case for RISC-V," Electrical Engineering and Computer Sciences, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, Aug. 2014.

[78] D. A. Patterson and J. L. Hennessy, Computer Organization and Design RISC-V Edition: The Hardware/Software Interface. Morgan Kaufmann, 2017.

[79] D. Patterson and A. Waterman, The RISC-V Reader: An Open Architecture Atlas. Berkeley, CA: Strawberry Canyon LLC, 2017.

[80] C. Böhm and G. Jacopini, "Flow diagrams, Turing machines and languages with only two formation rules," Commun. ACM, vol. 9, no. 5, pp. 366–371, May 1966, doi: 10.1145/365958.365999.

[81] A. La Gala, M. Chiariello, M. Malanchini, M. Tambaro and M. De Matteis, "Design and Test-Verification of a Single-Cycle RISC-V Microprocessor on FPGA," 2024 31st IEEE International Conference on Electronics, Circuits and Systems (ICECS), Nancy, France, 2024, pp. 1-4, doi: 10.1109/ICECS61496.2024.10848919.

[82] W. M. Elgharbawy, et al., "Leakage sources and possible solutions in nanometer CMOS technologies," in IEEE Circuits and Systems Magazine, vol. 5, no. 4, pp. 6-17, Fourth Quarter 2005, doi: 10.1109/MCAS.2005.1550165.

[83] Yin, et al., "Power, performance, and area evaluation across 180nm-28nm technology nodes based on benchmark circuits." IEICE Electron. Express 21 (2024): 20240194. https://doi.org/10.1587/elex.21.20240194.

[84] R. Saligram, et al., "Power Performance Analysis of Digital Standard Cells for 28 nm Bulk CMOS at Cryogenic Temperature Using BSIM Models," in IEEE

Journal on Exploratory Solid-State Computational Devices and Circuits, vol. 7, no. 2, pp. 193-200, Dec. 2021, doi: 10.1109/JXCDC.2021.3131100.

[85] Riscv-Software-Src. (n.d.). RISCV-software-src/RISCV-tests. GitHub. https://github.com/riscv-software-src/riscv-tests.

[86] Tyagi, et al., "TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities." ArXiv abs/2201.09941 (2022).

[87] C. K. Jha, et al., "cecApprox: Enabling Automated Combinational Equivalence Checking for Approximate Circuits," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 71, no. 7, pp. 3282-3293, July 2024, doi: 10.1109/TCSI.2024.3388256.

[88] H. Mohanty, et al., "Formal That "Floats" High: Formal Verification of Floating Point Arithmetic," 2025 37th International Conference on Microelectronics (ICM), Cairo, Egypt, 2025, pp. 1-6, doi: 10.1109/ICM66518.2025.11321329.

[89] R. Garcia-Ramirez et al., "Siwa: a RISC-V RV32I based Micro-Controller for Implantable Medical Applications," 2020 IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS), San Jose, Costa Rica, 2020, pp. 1-4, doi: 10.1109/LASCAS45839.2020.9068952.

[90] C. Duran, et al., "A system-onchip platform for the internet of things featuring a 32-bit risc-v based microcontroller," in 2017 IEEE 8th Latin American Symposium on Circuits Systems (LASCAS), Feb 2017, pp. 1–4.

[91] C. Duran et al., "A 32-bit RISC-V AXI4-lite bus-based microcontroller with 10-bit SAR ADC," 2016 IEEE 7th Latin American Symposium on Circuits & Systems (LASCAS), Florianopolis, Brazil, 2016, pp. 315-318, doi: 10.1109/LASCAS.2016.7451073.

[92] P. Davide Schiavone et al., "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications," 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS), Thessaloniki, Greece, 2017, pp. 1-8, doi: 10.1109/PATMOS.2017.8106976.

[93] A. K. Pires,et al., "Physical Implementation of the NoX RISC-V Core Using an Open PDK," 2025 38th SBC/SBMicro/IEEE Symposium on Integrated Circuits and Systems Design (SBCCI), Manaus, Brazil, 2025, pp. 1-5, doi: 10.1109/SBCCI66862.2025.11218642.

[94] I. Thanga Dharsni, et al., "Optimized Hazard Free Pipelined Architecture Block for RV32I RISC-V Processor," 2022 3rd International Conference on Smart Electronics and Communication (ICOSEC), Trichy, India, 2022, pp. 739-746, doi: 10.1109/ICOSEC54921.2022.9952122.

[95] S. Bora and R. Paily, "A High-Performance Core Micro-Architecture Based on RISC-V ISA for Low Power Applications," in IEEE Transactions on Circuits

and Systems II: Express Briefs, vol. 68, no. 6, pp. 2132-2136, June 2021, doi: 10.1109/TCSII.2020.3043204.

[96] M. Gautschi et al., "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 10, pp. 2700-2713, Oct. 2017, doi: 10.1109/TVLSI.2017.2654506.

[97] M. Gautschi et al., "Tailoring instruction-set extensions for an ultra-low power tightly-coupled cluster of OpenRISC cores," 2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), Daejeon, Korea (South), 2015, pp. 25-30, doi: 10.1109/VLSI-SoC.2015.7314386.

[98] M. Tambaro et al., "500 MHz CMOS 28-nm Floating Point Arithmetic Unit for 32-Bit RISC-V Microprocessors," 2024 19th Conference on Ph.D Research in Microelectronics and Electronics (PRIME), Larnaca, Cyprus, 2024, pp. 1-4, doi: 10.1109/PRIME61930.2024.10559675.

Appendix A

Verilog Source Code of RV32I Microprocessor

Source Code Files Hierarchy

```
top_core_serIO.v            //Top-Level System-on-Chip
|-- debug_serIO.v
|-- data_ram.v              // Data-Memory
|-- GPIO_serIO.v            // I/O-Module
|-- top_rv32i.v
|   |-- code_ram.v          // Code-Memory
|   |-- debug_module.v
|   |-- data_ram_mux.v
|   |-- code_ram_mux.v
|   |-- gpo_controller.v
|   |-- rv32i_core.v        // RV32I-Core
|   |   |-- decode.v
|   |   |-- register_file.v
|   |   |-- alu_mux.v
|   |   |-- alu.v
|   |   |-- fetch.v
|   |   |-- memory_controller.v
|   |   |-- pc_sel.v
|   |   |-- writeback.v
|   |   |-- cs_registers.v
|   |   |-- interrupt_mux.v
```

Parameters.v

```
`define MRET           32'b00110000001000000000000001110011
`define RET            32'b1000000001100111 // JALR instruction with rd=x0, imm=0, rs1=x1

// Instruction Format Type classification
`define UNDEFINED_TYPE 3'b000
`define R_TYPE         3'b001
`define I_TYPE         3'b010
`define S_TYPE         3'b011
`define B_TYPE         3'b100
`define U_TYPE         3'b101
`define J_TYPE         3'b110
`define OTHER_TYPE     3'b111

// FUNCT3 Field Field Definitions
`define F3_BEQ            3'b000
`define F3_BNE            3'b001
`define F3_BLT            3'b100
`define F3_BGE            3'b101
`define F3_BLTU           3'b110
`define F3_BGEU           3'b111
`define F3_ADD            3'b000
`define F3_SUB            3'b000
`define F3_SLL            3'b001
`define F3_SLT            3'b010
`define F3_SLTU           3'b011
```

```
`define F3_XOR            3'b100
`define F3_SRL            3'b101
`define F3_SRA            3'b101
`define F3_OR             3'b110
`define F3_AND            3'b111
`define F3_RW             3'b001
`define F3_RS             3'b010
`define F3_RC             3'b011
`define F3_RWI            3'b101
`define F3_RSI            3'b110
`define F3_RCI            3'b111
`define F3_MUL            3'b000
`define `F3_MULH          3'b001
`define F3_MULHSU         3'b010
`define F3_MULHU          3'b011
`define F3_DIV            3'b100
`define F3_DIVU           3'b101
`define F3_REM            3'b110
`define F3_REMU           3'b111

// Memory transfer size constraints (co-dependent with the Memory Controller sub-module configuration)
`define MEM_B             2'b00
`define MEM_H             2'b01
`define MEM_W             2'b10
`define MEM_INV           2'b11

// Multiplexer Operand Selection Codes
`define V1_RV1          1'b0
`define V1_PC           1'b1
`define V2_RV2          1'b0
`define V2_IMM          1'b1
`define WB_PC           2'b00
`define WB_MEM          2'b01
`define WB_ALU          2'b10
`define WB_IMM          2'b11

// Instruction Opcode Definitions
`define OP_LUI            7'b0110111
`define OP_AUIPC          7'b0010111
`define OP_JAL            7'b1101111
`define OP_JALR           7'b1100111
`define OP_BRANCH         7'b1100011
`define OP_LOAD           7'b0000011
`define OP_STORE          7'b0100011
`define OP_OPIMM          7'b0010011
`define OP_OP             7'b0110011
`define OP_SYSTEM         7'b1110011
`define OP_FENCE          7'b0001111

// Arithmetic Logic Unit (ALU) Operation Codes
`define ALU_BEQ            5'b00000
`define ALU_BNE            5'b00001
`define ALU_BLT            5'b00010
`define ALU_BGE            5'b00011
`define ALU_BLTU           5'b00100
`define ALU_BGEU           5'b00101
`define ALU_ADD            5'b00110
`define ALU_SUB            5'b00111
`define ALU_SLL            5'b01000
`define ALU_SLT            5'b01001
`define ALU_SLTU           5'b01010
`define ALU_XOR            5'b01011
`define ALU_SRL            5'b01100
`define ALU_SRA            5'b01101
`define ALU_OR             5'b01110
`define ALU_AND            5'b01111
`define ALU_MUL            5'b10000
`define ALU_MULH           5'b10001
`define ALU_MULHSU         5'b10010
`define ALU_MULHU          5'b10011
`define ALU_DIV            5'b10100
`define ALU_DIVU           5'b10101
`define ALU_REM            5'b10110
`define ALU_REMU           5'b10111
`define ALU_INV            5'b11111

// Control and Status Registers (CSR) Address Mappings
`define MSTATUS            12'h300
`define MISA               12'h301
`define MIE                12'h304
`define MTVEC              12'h305
`define MEPC               12'h341
`define MCAUSE             12'h342
`define MIP                12'h344

// Memory Map Space Allocations
`define DR_MEM              2'b00
```

```
`define CR_MEM              2'b01
`define REG_MEM             2'b10
`define IO_MEM              2'b11

`define DR_MEM  2'b00
`define CR_MEM  2'b01
`define REG_MEM  2'b10
`define IO_MEM  2'b11
```

top_core_serIO.v

```
module top_core_serIO
        #(
    parameter ADDR_WIDTH = 7,
    parameter GPO_NUM = 8,
    parameter GPO_ADDR_WIDTH = 2
        )(
    input wire en,
    input wire core_en,
    input wire clk,
    input wire nrst,
    input wire external_interrupt_top_i,
    output wire is_ret_top_o,
    input wire clk_serial,
    input wire serial_i,
    output wire serial_o,
    output wire serial_rdy_o,
    output wire [GPO_NUM-1:0] gpo_o,
    input wire gp_clk_serial_in_i,
    input wire gp_data_wr_i,
    input wire gp_serial_in_i,
    input wire gp_clk_serial_out_i,
    output wire gp_serial_out_o,
    input wire data_rd_i
  );

  // Additional precaution to better square the waveform
  CKBD8BWP35P140LVT ck_buff(.I(clk), .Z(clk_i));
  CKBD8BWP35P140LVT ck_buff_serial(.I(clk_serial), .Z(clk_serial_i));

   reg nrst_o;
  // RESET FLIP-FLOP
  always @(posedge clk_i) begin
    // reset
      nrst_o <= nrst;
  end
  // SERIALIZER SIGNALS
  wire [31:0] data_deb_to_out;
  wire data_ready;
  wire debug_mode;
  wire [1:0] mem_type;
  wire [ADDR_WIDTH-1:0] address_ser;
  wire [31:0] data_out_to_deb;
  debug_serIO #(
    .ADDR_WIDTH(ADDR_WIDTH)
  ) ds (
    // From out
    .nrst_i(nrst_o),
    .clk_i(clk_i),
    .clk_serial_i(clk_serial_i),
    .din_sipo_i(serial_i),
    // To out
    .dout_sipo_o(serial_o),
    .dout_rdy_o(serial_rdy_o), // synch to clk_i
    // From top
    .data_deb_to_out_i(data_deb_to_out),
    .data_ready_i(data_ready),
    // To top
    .debug_mode_o(debug_mode),
    .mem_type_o(mem_type),
    .address_o(address_ser),
    .data_out_to_deb_o(data_out_to_deb)
  );

  // DATA RAM SIGNALS
  wire [ADDR_WIDTH-1:0] addra_dr;
  wire [31:0] dina_dr;
  wire [3:0] mem_mask_dr;
  wire wea_dr;
  wire ena_dr;
  wire [31:0] douta_dr;
```

```
  data_ram #(.ADDR_WIDTH(ADDR_WIDTH)
  ) dr(
    .addra_i(addra_dr),
    .dina_i(dina_dr),
    .mem_mask_i(mem_mask_dr),
    .clk_i(clk_i),
    .wea_i(wea_dr),
    .ena_i(ena_dr),
    .douta_o(douta_dr)
        );
    wire we_from_out;
    wire [31:0] data_from_serial_i;
    wire [31:0] data_to_serial_o;
    wire [31:0] data_to_out_t;

  top_rv32i #(
    .ADDR_WIDTH(ADDR_WIDTH),
    .GPO_NUM(GPO_NUM),
    .GPO_ADDR_WIDTH(GPO_ADDR_WIDTH)
  ) top_rv(
    .en(en),
    .core_en(core_en),
    .clk(clk_i),
    .nrst(nrst_o),
    .external_interrupt_top_i(external_interrupt_top_i),
    .is_ret_top_o(is_ret_top_o),
    .gpo_o(gpo_o),
    .ena_dr_o(ena_dr),
    .wr_en_dr_o(wea_dr),
    .addra_dr_o(addra_dr),
    .dina_dr_o(dina_dr),
    .mem_mask_dr_o(mem_mask_dr),
    .douta_dr_i(douta_dr),
    .address_i(address_ser),
    .mem_type_i(mem_type),
    .debug_mode_i(debug_mode),
    .data_from_out_top(data_out_to_deb),
    .data_to_out_top(data_to_out_t),
    .data_ready_o(data_ready),
    .we_from_out(we_from_out),
    .data_from_serial_i(data_from_serial_i),
    .data_to_serial_o(data_to_serial_o)
  );
  // GPIO SERIO
  GPIO_serIO gios(
    .clk_i(clk_i),
    .nrst_i(nrst_o),
    .clk_serial_in_i(gp_clk_serial_in_i),
    .data_wr_i(gp_data_wr_i),
    .data_from_serial_o(data_from_serial_i),
    .data_wr_o(we_from_out),
    .clk_serial_out_i(gp_clk_serial_out_i),
    .data_to_serial_i(data_to_serial_o),
    .data_rd_i(data_rd_i),
    .serial_in_i(gp_serial_in_i),
    .serial_out_o(gp_serial_out_o)
  );
assign data_deb_to_out = data_to_out_t;
endmodule
```

debug_serIO.v

```
module debug_serIO #(
  parameter ADDR_WIDTH = 7
  )(
  // From out
  input nrst_i,
  input clk_i,
  input clk_serial_i,
  input din_sipo_i,
  // To out
  output dout_sipo_o,
  output dout_rdy_o, // synch to clk_i
  // From top
  input [31:0] data_deb_to_out_i,
  input data_ready_i,
  // To top
  output debug_mode_o,
  output [1:0] mem_type_o,
  output [ADDR_WIDTH-1:0] address_o,
```

```
  output [31:0] data_out_to_deb_o
);

  reg [ADDR_WIDTH+32:0] cmd_sipo_reg;
  // Read serial data input
  assign debug_mode_o = cmd_sipo_reg[0];
  assign mem_type_o = cmd_sipo_reg[2:1];
  // Re-align address bits by restoring the two previously truncated LSBs/MSBs based on memory space
     classification
  assign address_o = (mem_type_o == 2'b10)? {2'b0,cmd_sipo_reg[ADDR_WIDTH:3]} : {cmd_sipo_reg[
     ADDR_WIDTH:3],2'b0}; // Re-add the 2 removed bits
  assign data_out_to_deb_o = cmd_sipo_reg[ADDR_WIDTH+32:ADDR_WIDTH+1];
  reg data_ready_slow; // Synchronized data_ready signal asserted in the clk_i fast clock domain
  reg data_ready_ack;  // Acknowledge handshake signal asserted in the clk_serial slow clock domain

  integer i;
  integer j;
  always @(posedge clk_serial_i, negedge nrst_i) begin
    if (!nrst_i) begin
      cmd_sipo_reg <= 0;
      data_ready_ack <= 1'b0;
    end
    else begin
      cmd_sipo_reg[ADDR_WIDTH+32] <= din_sipo_i;
      if (!data_ready_slow)
        for (i = 7; i < ADDR_WIDTH+32; i = i+1)
          cmd_sipo_reg[i] <= cmd_sipo_reg[i+1];
      else begin
        cmd_sipo_reg[ADDR_WIDTH+32-1:ADDR_WIDTH+1-1] <= data_deb_to_out_i;
        data_ready_ack <= 1'b1;
      end
      for (j = 0; j < 7; j = j+1)
        cmd_sipo_reg[j] <= cmd_sipo_reg[j+1];
      if (data_ready_ack)
        data_ready_ack <= 1'b0;
    end
  end

  assign dout_sipo_o = cmd_sipo_reg[0];

  // Write parallel data output
  always @(posedge clk_i, negedge nrst_i) begin
    if (!nrst_i) begin
      data_ready_slow <= 1'b0;
    end
    else begin
      if (data_ready_i) // Propagate data_ready into the clk_i fast clock domain
        data_ready_slow <= 1'b1; // Synchronous assertion in the clk_i domain
      if (data_ready_ack) // Clear handshake latch upon detection of slow-domain acknowledge
        data_ready_slow <= 1'b0; // Synchronous deassertion in the clk_i domain
    end
  end

  assign dout_rdy_o = data_ready_slow;
endmodule
```

data_ram.v

```
module data_ram#( parameter ADDR_WIDTH = 7)(
  input [ADDR_WIDTH-1:0] addra_i,  // Address bus (must be word-aligned, i.e., terminating with 2'b00)
  input [31:0] dina_i,             // RAM input data bus
  input [3:0] mem_mask_i,          // Byte-enable memory mask for selective write operations
  input clk_i,                     // System clock signal
  input wea_i,                     // Synchronous write enable signal
  input ena_i,                     // Synchronous read enable signal
  output [31:0] douta_o            // RAM output data bus
);

  localparam nrow = 2**ADDR_WIDTH;
  reg [7:0] BRAM [nrow-1:0];
  wire [ADDR_WIDTH-3:0] address = addra_i[ADDR_WIDTH-1:2];
  wire unconnected_wires = addra_i[1:0];
  reg [31:0] ram_data;

  // Synchronous write block with byte-masking capabilities
  always @(posedge clk_i) begin
    if (wea_i) begin
      if(mem_mask_i[0])
        BRAM[{address,2'b00}] <= dina_i[7:0];
      if(mem_mask_i[1])
        BRAM[{address,2'b01}] <= dina_i[15:8];
```

```
      if(mem_mask_i[2])
        BRAM[{address,2'b10}] <= dina_i[23:16];
      if(mem_mask_i[3])
        BRAM[{address,2'b11}] <= dina_i[31:24];
    end
  end

  // Continuous assignment for read operations with synchronous output gating
  assign douta_o = (ena_i == 1'b0)? 32'b0 : {BRAM[{address,2'b11}],BRAM[{address,2'b10}],BRAM[{address
     ,2'b01}],BRAM[{address,2'b00}]};
endmodule
```

top_rv32i.v

```
module top_rv32i
        #(
    parameter ADDR_WIDTH = 7,
    parameter GPO_NUM = 8,
    parameter GPO_ADDR_WIDTH = 2
        )(
    input wire en,
    input wire core_en,
    input wire clk,
    input wire nrst,
    input wire external_interrupt_top_i,
    output wire is_ret_top_o,
    // Debugger interface signals
    input wire [ADDR_WIDTH-1:0] address_i,
    input wire [1:0] mem_type_i,
    input wire debug_mode_i,
    input wire [31:0] data_from_out_top,
    output wire [31:0] data_to_out_top,
    output wire data_ready_o,
    // Data RAM interface signals
    output wire ena_dr_o,
    output wire wr_en_dr_o,
    output wire [31:0] dina_dr_o,
    output wire [3:0] mem_mask_dr_o,
    output wire [ADDR_WIDTH-1:0] addra_dr_o,
    input wire [31:0] douta_dr_i,
    // General Purpose Outputs (GPO) and Serial Link
    input wire we_from_out,
    output wire [GPO_NUM-1:0] gpo_o,
    input wire [31:0] data_from_serial_i,
    output wire [31:0] data_to_serial_o
        );

  // RISC-V CORE INTERNAL SIGNALS
  wire en_rv_i;
  wire clk_rv_i;
  wire nrst_rv_i;
  wire [31:0] out_pc_rv_i;
  wire wr_pc_rv_i;
  wire [4:0] reg_addr_rv_i;
  wire [31:0] rwr_deb_rv_i;
  wire wr_en_deb_rv_i;
  wire [31:0] reg_value_rv_o;
  wire external_interrupt_rv_i;
  wire is_ret_rv_o;
  wire [31:0] inst_rv_i;
  wire [ADDR_WIDTH-1:0] pc_rv_o;
  wire cren_rv_o;
  wire [31:0] din_rv_i;
  wire [31:0] dout_rv_o;
  wire [ADDR_WIDTH-1:0] addr_rv_o;
  wire [1:0] mem_type_rv_o;
  wire [3:0] mem_mask_rv_o;
  wire drwren_rv_o;
  wire dren_rv_o;

  rv32i_core #(
    .ADDR_WIDTH(ADDR_WIDTH)
  ) rv(
    .en_rv_i(en_rv_i),
    .clk_rv_i(clk_rv_i),
    .nrst_rv_i(nrst_rv_i),
    .out_pc_rv_i(out_pc_rv_i),
    .wr_pc_rv_i(wr_pc_rv_i),
    .reg_addr_rv_i(reg_addr_rv_i),
    .rwr_deb_rv_i(rwr_deb_rv_i),
    .wr_en_deb_rv_i(wr_en_deb_rv_i),
```

```
  .reg_value_rv_o(reg_value_rv_o),
  .external_interrupt_rv_i(external_interrupt_rv_i),
  .is_ret_rv_o(is_ret_rv_o),
  .inst_rv_i(inst_rv_i),
  .pc_rv_o(pc_rv_o),
  .cren_rv_o(cren_rv_o),
  .din_rv_i(din_rv_i),
  .dout_rv_o(dout_rv_o),
  .addr_rv_o(addr_rv_o),
  .mem_type_rv_o(mem_type_rv_o),
  .mem_mask_rv_o(mem_mask_rv_o),
  .drwren_rv_o(drwren_rv_o),
  .dren_rv_o(dren_rv_o)
);

// CODE RAM INTERNAL SIGNALS
wire [ADDR_WIDTH-1:0] addra_cr_i;
wire clk_cr_i;
wire ena_cr_i;
wire nrsta_cr_i;
wire wr_en_cr_i;
wire [31:0] dina_cr_i;
wire [31:0] douta_cr_o;

code_ram #(.ADDR_WIDTH(ADDR_WIDTH)) cr(
  .addra(addra_cr_i),
  .clk(clk_cr_i),
  .ena(ena_cr_i),
  .nrsta(nrsta_cr_i),
  .wr_en(wr_en_cr_i),
  .dina(dina_cr_i),
  .douta(douta_cr_o)
);

// DEBUG MODULE INTERNAL SIGNALS
wire en_dm_i;
wire clk_dm_i;
wire nrst_dm_i;
wire is_ret_core_dm_i;
wire core_en_dm_i;
wire debug_mode_dm_i;
wire [ADDR_WIDTH-1:0] address_dm_i;
wire [1:0] mem_type_dm_i;
wire [31:0] data_from_out_dm;
wire [31:0] data_to_out_dm;
wire data_ready_dm_o;
wire [31:0] data_to_int_dm;
wire [ADDR_WIDTH-1:0] address_int_dm_o;
wire [1:0] mem_type_int_dm_o;
wire core_en_dm_o;
wire sel_cr_dm_o;
wire wren_cr_dm_o;
wire [31:0] data_from_cr_dm;
wire sel_dr_dm_o;
wire wren_dr_dm_o;
wire [31:0] data_from_dr_dm;
wire wren_reg_dm_o;
wire [31:0] data_from_reg_dm;
wire wr_pc_dm_o;
wire [31:0] out_pc_dm_o;

debug_module #(
  .ADDR_WIDTH(ADDR_WIDTH)
) dm(
  .en_i(en_dm_i),
  .clk_i(clk_dm_i),
  .nrst_i(nrst_dm_i),
  .is_ret_core_i(is_ret_core_dm_i),
  .core_en_i(core_en_dm_i),
  .debug_mode_i(debug_mode_dm_i),
  .address_i(address_dm_i),
  .mem_type_i(mem_type_dm_i),
  .data_from_out_i(data_from_out_dm),
  .data_to_out_o(data_to_out_dm),
  .data_ready_o(data_ready_dm_o),
  .data_to_int_o(data_to_int_dm),
  .address_int_o(address_int_dm_o),
  .mem_type_int_o(mem_type_int_dm_o),
  .core_en_o(core_en_dm_o),
  .sel_cr_o(sel_cr_dm_o),
  .wren_cr_o(wren_cr_dm_o),
  .data_from_cr_i(data_from_cr_dm),
  .sel_dr_o(sel_dr_dm_o),
  .wren_dr_o(wren_dr_dm_o),
  .data_from_dr_i(data_from_dr_dm),
  .wren_reg_o(wren_reg_dm_o),
  .data_from_reg_i(data_from_reg_dm),
```

```
  .wr_pc_o(wr_pc_dm_o),
  .out_pc_o(out_pc_dm_o)
);

// DATA RAM MULTIPLEXER (MUX) SIGNALS
wire [ADDR_WIDTH-1:0] addr_rv_drm_i;          // Current Program Counter (PC)
wire [1:0] mem_type_rv_drm_i;
wire [31:0] dout_rv_drm_i;                    // Data value from the Register File
wire [3:0] mem_mask_rv_drm_i;                 // Byte-enable mask from Instruction Decoder
wire drwren_rv_drm_i;                         // Write enable routing selection from Decoder
wire en_rv_drm_i;
wire [31:0] data_from_dr_drm;
wire [31:0] data_from_gpo_drm;
wire [31:0] data_to_core_drm;
wire [ADDR_WIDTH-1:0] addr_deb_drm_i;
wire [1:0] mem_type_deb_drm_i;
wire [31:0] dout_deb_drm_i;                   // Debug value destined for the Register File
wire drwren_deb_drm_i;                        // Debug write enable routing selection
wire sel_deb_drm_i ;
wire [ADDR_WIDTH-1:0] addr_drm_o;
wire [31:0] dout_drm_o;                       // Output data value formatted for the Register File
wire [3:0] mem_mask_drm_o;                    // Multiplexed output byte mask
wire out_wren_drm_o;
wire out_en_drm_o;
wire drwren_drm_o;                            // Multiplexed Data RAM write enable signal
wire dren_drm_o;

data_ram_mux #(
  .ADDR_WIDTH(ADDR_WIDTH)
) drm(
  .addr_rv_i(addr_rv_drm_i),
  .mem_type_rv_i(mem_type_rv_drm_i),
  .dout_rv_i(dout_rv_drm_i),
  .mem_mask_rv_i(mem_mask_rv_drm_i),
  .drwren_rv_i(drwren_rv_drm_i),
  .en_rv_i(en_rv_drm_i),
  .data_from_dr_i(data_from_dr_drm),
  .data_from_gpo_i(data_from_gpo_drm),
  .data_to_core_o(data_to_core_drm),
  .addr_deb_i(addr_deb_drm_i),
  .mem_type_deb_i(mem_type_deb_drm_i),
  .dout_deb_i(dout_deb_drm_i),
  .drwren_deb_i(drwren_deb_drm_i),
  .sel_deb_i(sel_deb_drm_i),
  .addr_o(addr_drm_o),
  .dout_o(dout_drm_o),
  .mem_mask_o(mem_mask_drm_o),
  .out_wren_o(out_wren_drm_o),
  .out_en_o(out_en_drm_o),
  .drwren_o(drwren_drm_o),
  .dren_o(dren_drm_o)
);

// CODE RAM MULTIPLEXER (MUX) SIGNALS
wire [ADDR_WIDTH-1:0] addr_rv_crm_i;
wire en_rv_crm_i;
wire [ADDR_WIDTH-1:0] addr_deb_crm_i;
wire wren_deb_crm_i;
wire [31:0] data_from_deb_crm;
wire sel_cr_deb_crm_i;
wire [ADDR_WIDTH-1:0] addr_crm_o;
wire en_crm_o;
wire wren_crm_o;
wire [31:0] data_to_cr_crm;

code_ram_mux #(
  .ADDR_WIDTH(ADDR_WIDTH)
) crm(
  .addr_rv_i(addr_rv_crm_i),
  .en_rv_i(en_rv_crm_i),
  .addr_deb_i(addr_deb_crm_i),
  .wren_deb_i(wren_deb_crm_i),
  .data_from_deb(data_from_deb_crm),
  .sel_cr_deb_i(sel_cr_deb_crm_i),
  .addr_o(addr_crm_o),
  .en_o(en_crm_o),
  .wren_o(wren_crm_o),
  .data_to_cr_o(data_to_cr_crm)
);

// OUTPUT CONTROLLER SIGNALS
wire clk_gc;
wire nrst_gc;
wire en_gc_i;                             // Peripheral controller enable from Core
wire [ADDR_WIDTH-1:0] address_gc_i;       // Memory-mapped I/O address from ALU
wire wr_en_gc_i;
wire [3:0] mem_mask_gc_i;                 // Memory access type (Word, Halfword, Byte)
```

```
  wire [31:0] data_from_core_gc_i;         // Data payload to be stored from Register File
  wire [31:0] data_to_core_gc_o;           // Loaded data payload from peripheral to Register File
  wire [GPO_NUM-1:0] data_to_out_gc_o;
  wire we_from_out_gc;
  wire [31:0] data_from_serial_gc_i;
  wire [31:0] data_to_serial_gc_o;

  gpo_controller #(
    .ADDR_WIDTH(ADDR_WIDTH),
    .GPO_NUM(GPO_NUM),
    .GPO_ADDR_WIDTH(GPO_ADDR_WIDTH)
  ) gc(
    .clk(clk_gc),
    .nrst(nrst_gc),
    .en_i(en_gc_i),                            // Peripheral controller enable from Core
    .address_i(address_gc_i),                  // Memory-mapped I/O address from ALU
    .wr_en_i(wr_en_gc_i),
    .mem_mask_i(mem_mask_gc_i),                // Memory access type (Word, Halfword, Byte)
    .data_from_core_i(data_from_core_gc_i),    // Data payload to be stored from Register File
    .we_from_out(we_from_out_gc),
    .data_to_core_o(data_to_core_gc_o),        // Loaded data payload from peripheral to Register File
    .data_to_out_o(data_to_out_gc_o),
    .data_from_serial_i(data_from_serial_gc_i),
    .data_to_serial_o(data_to_serial_gc_o)
  );

  // RISC-V CORE PORT INTERCONNECTIONS
  assign en_rv_i = core_en_dm_o;
  assign clk_rv_i = clk;
  assign nrst_rv_i = nrst;
  assign out_pc_rv_i = out_pc_dm_o;
  assign wr_pc_rv_i = wr_pc_dm_o;
  assign reg_addr_rv_i = address_int_dm_o[4:0];
  assign rwr_deb_rv_i = data_to_int_dm;
  assign wr_en_deb_rv_i = wren_reg_dm_o;
  assign external_interrupt_rv_i = external_interrupt_top_i;
  assign is_ret_top_o = is_ret_rv_o;
  assign inst_rv_i = douta_cr_o;
  assign din_rv_i = data_to_core_drm;

  // CODE RAM PORT INTERCONNECTIONS
  assign addra_cr_i = addr_crm_o;
  assign clk_cr_i = clk;
  assign ena_cr_i = en_crm_o;
  assign nrsta_cr_i = nrst;
  assign wr_en_cr_i = wren_crm_o;
  assign dina_cr_i = data_to_cr_crm;

  // DATA RAM PORT INTERCONNECTIONS
  assign addra_dr_o = addr_drm_o[ADDR_WIDTH-1:0];
  assign dina_dr_o = dout_drm_o;
  assign mem_mask_dr_o = mem_mask_drm_o;
  assign wr_en_dr_o = drwren_drm_o;
  assign ena_dr_o = dren_drm_o;

  // DEBUG MODULE PORT INTERCONNECTIONS
  assign en_dm_i = en;
  assign clk_dm_i = clk;
  assign nrst_dm_i = nrst;
  assign is_ret_core_dm_i = is_ret_rv_o;
  assign core_en_dm_i = core_en;
  assign debug_mode_dm_i = debug_mode_i;
  assign address_dm_i = address_i;
  assign mem_type_dm_i = mem_type_i;
  assign data_from_out_dm = data_from_out_top;
  assign data_to_out_top = data_to_out_dm;
  assign data_ready_o = data_ready_dm_o;
  assign data_from_cr_dm = douta_cr_o;
  assign data_from_dr_dm = douta_dr_i;
  assign data_from_reg_dm = reg_value_rv_o;

  // DATA RAM MUX PORT INTERCONNECTIONS
  assign addr_rv_drm_i = addr_rv_o;
  assign mem_type_rv_drm_i = mem_type_rv_o;
  assign dout_rv_drm_i = dout_rv_o;
  assign mem_mask_rv_drm_i = mem_mask_rv_o;
  assign drwren_rv_drm_i = drwren_rv_o;
  assign en_rv_drm_i = dren_rv_o;
  assign data_from_dr_drm = douta_dr_i;
  assign data_from_gpo_drm = data_to_core_gc_o;
  assign addr_deb_drm_i = address_int_dm_o;
  assign mem_type_deb_drm_i = mem_type_int_dm_o;
  assign dout_deb_drm_i = data_to_int_dm;
  assign drwren_deb_drm_i = wren_dr_dm_o;
  assign sel_deb_drm_i  = sel_dr_dm_o;

  // CODE RAM MUX PORT INTERCONNECTIONS
```

```
  assign addr_rv_crm_i = pc_rv_o;
  assign en_rv_crm_i = cren_rv_o;
  assign addr_deb_crm_i = address_int_dm_o;
  assign wren_deb_crm_i = wren_cr_dm_o;
  assign data_from_deb_crm = data_to_int_dm;
  assign sel_cr_deb_crm_i = sel_cr_dm_o;

  // OUTPUT CONTROLLER PORT INTERCONNECTIONS
  assign clk_gc = clk;
  assign nrst_gc = nrst;
  assign en_gc_i = out_en_drm_o;
  assign address_gc_i = addr_drm_o;
  assign wr_en_gc_i = out_wren_drm_o;
  assign mem_mask_gc_i = mem_mask_drm_o;
  assign data_from_core_gc_i = dout_drm_o;
  assign gpo_o = data_to_out_gc_o;
  assign we_from_out_gc = we_from_out;
  assign data_from_serial_gc_i = data_from_serial_i;
  assign data_to_serial_o = data_to_serial_gc_o;

endmodule
```

GPIO_serIO.v

```
module GPIO_serIO(
  input clk_i,
  input nrst_i,
  input clk_serial_in_i,
  input data_wr_i,
  output [31:0] data_from_serial_o,
  output data_wr_o,
  input clk_serial_out_i,
  input [31:0] data_to_serial_i,
  input data_rd_i,
  input serial_in_i,
  output serial_out_o
);

  reg [31:0] serial_shift_reg_in;
  reg [31:0] serial_shift_reg_out;

  // Serial-to-Parallel (Input) Stream Processing
  integer ii;
  always @(posedge clk_serial_in_i, negedge nrst_i) begin
    if (!nrst_i) begin
      serial_shift_reg_in <= 0;
    end
    else begin
      serial_shift_reg_in[31] <= serial_in_i;
      for (ii=0; ii<31; ii=ii+1)
        serial_shift_reg_in[ii] <= serial_shift_reg_in[ii+1];
      end
  end

  // Synchronous Edge Detection Logic for Write Enable Signal
  reg data_wr_prev;
  always @(posedge clk_i, negedge nrst_i) begin
    if (!nrst_i) begin
      data_wr_prev <= 1'b0;
    end
    else begin
      data_wr_prev <= data_wr_i;
    end
  end
  assign data_wr_o = !data_wr_prev & data_wr_i;
  assign data_from_serial_o = serial_shift_reg_in;

  // Parallel-to-Serial (Output) Stream Processing:
  // - If the data read signal is asserted concurrently with the clock edge, sample the input parallel
     word.
  // - If only the clock edge is asserted, perform a logical right-shift operation.
  integer io;
  always @(posedge clk_serial_out_i, negedge nrst_i) begin
    if (!nrst_i) begin
      serial_shift_reg_out <= 0;
    end
    else begin
      if (data_rd_i)
        serial_shift_reg_out <= data_to_serial_i;
      else begin
        for (io=0; io<31; io=io+1)
```

```
          serial_shift_reg_out[io] <= serial_shift_reg_out[io+1];
      end
    end
  end
  assign serial_out_o = serial_shift_reg_out[0];

endmodule
```

rv32i_core.v

```
module rv32i_core #(parameter ADDR_WIDTH = 7)(
    input wire en_rv_i,                     // Core enable signal used for stalling logic
    input wire clk_rv_i,                    // System clock signal
    input wire nrst_rv_i,                   // Asynchronous active-low reset signal
    // DEBUG INTERFACE
    input wire [31:0] out_pc_rv_i,          // Program Counter (PC) forced from the external DEBUG
     module
    input wire wr_pc_rv_i,                  // Control signal asserted when the PC is overridden from
     an external source
    input wire [4:0] reg_addr_rv_i,         // Register File address designated for debug operations
    input wire [31:0] rwr_deb_rv_i,         // Data payload to be written into the Register File from
     the DEBUG module
    input wire wr_en_deb_rv_i,              // Write enable signal asserted by the external Debug
     module
    output wire [31:0] reg_value_rv_o,      // Data value read from the Register File and routed to the
      Debug module
    input wire external_interrupt_rv_i,     // Asynchronous external interrupt request signal
    output wire is_ret_rv_o,
    // CODE RAM INTERFACE
    input wire [31:0] inst_rv_i,            // Fetch instruction word received from the CODE RAM
    output wire [ADDR_WIDTH-1:0] pc_rv_o,   // Program Counter bus routed from the Core to the Code RAM
    output wire cren_rv_o,                  // CODE RAM chip enable control signal
    // DATA RAM INTERFACE
    input wire [31:0] din_rv_i,             // Input data payload read from the DATA RAM to the Core
    output wire [31:0] dout_rv_o,           // Output data payload written from the Core to the DATA
     RAM
    output wire [ADDR_WIDTH-1:0] addr_rv_o, // Target memory address bus routed to the DATA RAM
    output wire [1:0] mem_type_rv_o,
    output wire [3:0] mem_mask_rv_o,        // Byte-enable memory mask for selective DATA RAM access
    output wire drwren_rv_o,                // DATA RAM synchronous write enable signal
    output wire dren_rv_o);                 // DATA RAM synchronous read enable signal

    ////////////////////////////////////////////////////////////////////////////////
    // MODULE INSTANTIATIONS
    ////////////////////////////////////////////////////////////////////////////////

    // INSTRUCTION FETCH (IF) STAGE SIGNALS
    wire en_f_i;
    wire clk_f_i;
    wire nrst_f_i;
    wire [31:0] next_pc_f_i;
    wire read_en_f_o;
    wire [31:0] pc_f_o;
    wire instr_misaligned_f_o;

    fetch f(
    .en_i(en_f_i),
    .clk_i(clk_f_i),
    .nrst_i(nrst_f_i),
    .next_pc_i(next_pc_f_i),
    .read_en_o(read_en_f_o),
    .pc_o(pc_f_o),
    .instr_misaligned_o(instr_misaligned_f_o));
    ////////////////////////////////////////////////////////////////////////////////

    // INSTRUCTION DECODE (ID) STAGE SIGNALS
    wire [31:0] instr_d_i;
    wire [4:0] rd_d_o;
    wire [4:0] rs1_d_o;
    wire [4:0] rs2_d_o;
    wire [2:0] func3_d_o;
    wire [31:0] immediate_d_o;
    wire is_csr_d_o;
    wire rd_we_d_o;
    wire [4:0] alu_op_d_o;
    wire v1sel_d_o;
    wire v2sel_d_o;
    wire [1:0] wbsel_d_o;
    wire is_load_d_o;
    wire is_store_d_o;
    wire [1:0] mem_size_d_o;
```

```
    wire uload_d_o;
    wire is_jalr_d_o;
    wire is_jump_d_o;
    wire is_mret_d_o;
    wire is_ret_d_o;
    wire illegal_instr_d_o;
    wire system_instr_d_o;

    decode d(
    .instr_i(instr_d_i),
    .rd_o(rd_d_o),
    .rs1_o(rs1_d_o),
    .rs2_o(rs2_d_o),
    .func3_o(func3_d_o),
    .immediate_o(immediate_d_o),
    .is_csr_o(is_csr_d_o),
    .rd_we_o(rd_we_d_o),
    .alu_op_o(alu_op_d_o),
    .v1sel_o(v1sel_d_o),
    .v2sel_o(v2sel_d_o),
    .wbsel_o(wbsel_d_o),
    .is_load_o(is_load_d_o),
    .is_store_o(is_store_d_o),
    .mem_size_o(mem_size_d_o),
    .uload_o(uload_d_o),
    .is_jalr_o(is_jalr_d_o),
    .is_jump_o(is_jump_d_o),
    .is_mret_o(is_mret_d_o),
    .is_ret_o(is_ret_d_o),
    .illegal_instr_o(illegal_instr_d_o),
    .system_instr_o(system_instr_d_o));
    ////////////////////////////////////////////////////////////////////////////////

    // REGISTER FILE (RF) SIGNALS
    wire en_rf_i;
    wire clk_rf_i;
    wire nrst_rf_i;
    wire [4:0] rs1_rf_i;
    wire [4:0] rs2_rf_i;
    wire [4:0] rd_rf_i;
    wire wr_en_rf_i;
    wire [31:0] rd_value_rf_i;
    wire wr_en_deb_rf_i;
    wire [4:0] rsdeb_rf_i;
    wire [31:0] rwr_deb_rf_i;
    wire [31:0] rvdeb_rf_o;
    wire [31:0] rv1_rf_o;
    wire [31:0] rv2_rf_o;

    register_file rf(
    .en(en_rf_i),
    .clk(clk_rf_i),
        .nrst(nrst_rf_i),
        .rs1_i(rs1_rf_i),
        .rs2_i(rs2_rf_i),
        .rd_i(rd_rf_i),
    .wr_en_i(wr_en_rf_i),
        .rd_value_i(rd_value_rf_i),
        .wr_en_deb_i(wr_en_deb_rf_i),
        .rsdeb_i(rsdeb_rf_i),
        .rwr_deb_i(rwr_deb_rf_i),
        .rvdeb_o(rvdeb_rf_o),
        .rv1_o(rv1_rf_o),
        .rv2_o(rv2_rf_o));
    ////////////////////////////////////////////////////////////////////////////////

    // ARITHMETIC LOGIC UNIT OPERAND MULTIPLEXER (ALU MUX) SIGNALS
    wire [31:0] pc_am_i;
    wire [31:0] rv1_am_i;
    wire v1_sel_am_i;
    wire [31:0] rv2_am_i;
    wire [31:0] imm_am_i;
    wire v2_sel_am_i;
    wire [31:0] value1_am_o;
    wire [31:0] value2_am_o;

     alu_mux am(
    .pc_i(pc_am_i),
        .rv1_i(rv1_am_i),
        .v1_sel_i(v1_sel_am_i),
        .rv2_i(rv2_am_i),
        .imm_i(imm_am_i),
    .v2_sel_i(v2_sel_am_i),
        .value1_o(value1_am_o),
        .value2_o(value2_am_o));
    ////////////////////////////////////////////////////////////////////////////////
```

```
    // ARITHMETIC LOGIC UNIT (ALU) SIGNALS
    wire [31:0] value1_a_i;
    wire [31:0] value2_a_i;
    wire [31:0] immediate_a_i;
    wire [31:0] pc_a_i;
    wire [4:0] alu_op_a_i;
    wire is_jalr_a_i;
    wire [31:0] adder_result_a_o;
    wire branch_result_a_o;
    wire [31:0] result_a_o;
    wire [31:0] next_pc_a_o;

     alu a(
    .value1_i(value1_a_i),
        .value2_i(value2_a_i),
        .immediate_i(immediate_a_i),
        .pc_i(pc_a_i),
        .alu_op_i(alu_op_a_i),
    .is_jalr_i(is_jalr_a_i),
        .adder_result_o(adder_result_a_o),
        .branch_result_o(branch_result_a_o),
        .result_o(result_a_o),
        .next_pc_o(next_pc_a_o));
    ////////////////////////////////////////////////////////////////////////////////

    // MEMORY CONTROLLER SIGNALS
    wire en_mc_i;
    wire [31:0] address_mc_i;
    wire [1:0] mem_size_mc_i;
    wire uload_mc_i;
    wire [31:0] store_value_mc_i;
    wire [31:0] load_value_mc_i;
    wire is_load_mc_i;
    wire is_store_mc_i;
    wire [31:0] store_value_mc_o;
    wire [31:0] load_value_mc_o;
    wire wr_en_mc_o;
    wire rd_en_mc_o;
    wire [3:0] mem_mask_mc_o;
    wire not_aligned_load_mc_o;
    wire not_aligned_store_mc_o;
    wire [ADDR_WIDTH-1:0] aligned_addr_mc_o;
    wire [1:0] mem_type_mc_o;

    memory_controller #(.ADDR_WIDTH(ADDR_WIDTH)) mc(
    .en_i(en_mc_i),
    .address_i(address_mc_i),
        .mem_size_i(mem_size_mc_i),
        .uload_i(uload_mc_i),
        .store_value_i(store_value_mc_i),
        .load_value_i(load_value_mc_i),
    .is_load_i(is_load_mc_i),
        .is_store_i(is_store_mc_i),
        .store_value_o(store_value_mc_o),
        .load_value_o(load_value_mc_o),
        .wr_en_o(wr_en_mc_o),
    .rd_en_o(rd_en_mc_o),
        .mem_mask_o(mem_mask_mc_o),
        .not_aligned_load_o(not_aligned_load_mc_o),
        .not_aligned_store_o(not_aligned_store_mc_o),
        .aligned_addr_o(aligned_addr_mc_o),
    .mem_type_o(mem_type_mc_o));
    ////////////////////////////////////////////////////////////////////////////////

    // PROGRAM COUNTER MULTIPLEXER (PC MUX) SIGNALS
    wire [31:0] pc_pm_i;
    wire [31:0] pc_alu_pm_i;
    wire [31:0] out_pc_pm_i;
    wire is_jump_pm_i;
    wire branch_result_pm_i;
    wire sel_pc_pm_i;
    wire [31:0] pc_pm_o;
    wire [31:0] pc4_pm_o;

    pc_sel pm(
    .pc_i(pc_pm_i),
        .pc_alu_i(pc_alu_pm_i),
        .out_pc_i(out_pc_pm_i),
        .is_jump_i(is_jump_pm_i),
        .branch_result_i(branch_result_pm_i),
    .sel_pc_i(sel_pc_pm_i),
    .pc_o(pc_pm_o),
    .pc4_o(pc4_pm_o));
    ////////////////////////////////////////////////////////////////////////////////

    // WRITEBACK MULTIPLEXER (WB MUX) SIGNALS
    wire [31:0] pc_wm_i;
```

```
    wire [31:0] mem_wm_i;
    wire [31:0] alu_wm_i;
    wire [31:0] imm_wm_i;
    wire [31:0] csr_wm_i;
    wire [1:0] wbsel_wm_i;
    wire [31:0] rd_value_wm_o;
    wire is_csr_wm_i;

    writeback wm(
    .pc_i(pc_wm_i),
        .mem_i(mem_wm_i),
        .alu_i(alu_wm_i),
        .imm_i(imm_wm_i),
        .csr_i(csr_wm_i),
        .wbsel_i(wbsel_wm_i),
        .is_csr_i(is_csr_wm_i),
    .rd_value_o(rd_value_wm_o));
    ////////////////////////////////////////////////////////////////////////////////

    // CONTROL AND STATUS REGISTERS (CSR) SIGNALS
    wire en_csr;
    wire clk_csr;
    wire nrst_csr;
    wire is_csr_csr_i;
    wire [2:0] func3_csr_i;
    wire [31:0] csr_index_csr_i;      // Shares the same routing wire as the immediate vector from the
     decoder
    wire [4:0] immediate_csr_i;       // Shares the same routing wire as the rs1 descriptor from the
     decoder
    wire [31:0] rv1_csr_i;
    wire [31:0] csr_rd_csr_o;
    wire external_interrupt_csr_i;    // Propagated external interrupt request line
    wire illegal_instr_csr_i;
    wire system_instr_csr_i;
    wire instr_misaligned_csr_i;
    wire misaligned_load_csr_i;
    wire misaligned_store_csr_i;
    wire [31:0] pc_csr_i;
    wire is_mret_csr_i;
    wire enter_trap_csr_o;
    wire exit_trap_csr_o;
    wire [31:0] return_address_csr_o;
    wire [31:0] trap_address_csr_o;

    cs_registers csr(
    .en(en_csr),
    .clk(clk_csr),
    .nrst(nrst_csr),
    .is_csr_i(is_csr_csr_i),
    .func3_i(func3_csr_i),
    .csr_index_i(csr_index_csr_i), // Shares the same routing wire as the immediate vector from the
     decoder
    .immediate_i(immediate_csr_i), // Shares the same routing wire as the rs1 descriptor from the
     decoder
    .rv1_i(rv1_csr_i),
    .csr_rd_o(csr_rd_csr_o),
    .external_interrupt_i(external_interrupt_csr_i),  // Propagated external interrupt request line
    .illegal_instr_i(illegal_instr_csr_i),
    .system_instr_i(system_instr_csr_i),
    .instr_misaligned_i(instr_misaligned_csr_i),
    .misaligned_load_i(misaligned_load_csr_i),
    .misaligned_store_i(misaligned_store_csr_i),
    .pc_i(pc_csr_i),
    .is_mret_i(is_mret_csr_i),
    .enter_trap_o(enter_trap_csr_o),
    .exit_trap_o(exit_trap_csr_o),
    . return_address_o(return_address_csr_o),
    . trap_address_o(trap_address_csr_o));
    ////////////////////////////////////////////////////////////////////////////////

    // INTERRUPT MULTIPLEXER (INT MUX) SIGNALS
    wire [31:0] cur_pc_im_i;
    wire [31:0] pc_trap_im_i;
    wire [31:0] pc_prev_im_i;
    wire enter_trap_im_i;
    wire exit_trap_im_i;
    wire sel_out_pc_im_i;
    wire [31:0] pc_im_o;

    interrupt_mux im(
    .cur_pc_i(cur_pc_im_i),
    .pc_trap_i(pc_trap_im_i),
    .pc_prev_i(pc_prev_im_i),
    .enter_trap_i(enter_trap_im_i),
    .exit_trap_i(exit_trap_im_i),
    .sel_out_pc_i(sel_out_pc_im_i),
    .pc_o(pc_im_o));
```

```
////////////////////////////////////////////////////////////////////////////////
// PORT INTERCONNECTIONS AND ASSIGNMENTS
////////////////////////////////////////////////////////////////////////////////

// INSTRUCTION FETCH STAGE INTERCONNECTIONS
assign en_f_i = en_rv_i;
assign clk_f_i = clk_rv_i;
assign nrst_f_i = nrst_rv_i;
assign next_pc_f_i = pc_im_o;

// INSTRUCTION DECODE STAGE INTERCONNECTIONS
assign instr_d_i = inst_rv_i;

// REGISTER FILE INTERCONNECTIONS
assign en_rf_i = en_rv_i;
assign clk_rf_i = clk_rv_i;
assign nrst_rf_i = nrst_rv_i;
assign rs1_rf_i = rs1_d_o;
assign rs2_rf_i = rs2_d_o;
assign rd_rf_i = rd_d_o;
assign wr_en_rf_i = rd_we_d_o;
assign rd_value_rf_i = rd_value_wm_o;
assign wr_en_deb_rf_i = wr_en_deb_rv_i;
assign rsdeb_rf_i = reg_addr_rv_i;
assign rwr_deb_rf_i = rwr_deb_rv_i;

// ALU MULTIPLEXER INTERCONNECTIONS
assign pc_am_i = pc_f_o;
assign rv1_am_i = rv1_rf_o;
assign v1_sel_am_i = v1sel_d_o;
assign rv2_am_i = rv2_rf_o;
assign imm_am_i = immediate_d_o;
assign v2_sel_am_i = v2sel_d_o;

// ARITHMETIC LOGIC UNIT INTERCONNECTIONS
assign value1_a_i = value1_am_o;
assign value2_a_i = value2_am_o;
assign immediate_a_i = immediate_d_o;
assign pc_a_i = pc_f_o;
assign alu_op_a_i = alu_op_d_o;
assign is_jalr_a_i = is_jalr_d_o;

// MEMORY CONTROLLER INTERCONNECTIONS
assign en_mc_i = en_rv_i;
assign address_mc_i = adder_result_a_o;
assign mem_size_mc_i = mem_size_d_o;
assign uload_mc_i = uload_d_o;
assign store_value_mc_i = rv2_rf_o;
assign load_value_mc_i = din_rv_i;
assign is_load_mc_i = is_load_d_o;
assign is_store_mc_i = is_store_d_o;

// PROGRAM COUNTER MULTIPLEXER INTERCONNECTIONS
assign pc_pm_i = pc_f_o;
assign pc_alu_pm_i = next_pc_a_o;
assign out_pc_pm_i = out_pc_rv_i;
assign is_jump_pm_i = is_jump_d_o;
assign branch_result_pm_i = branch_result_a_o;
assign sel_pc_pm_i = wr_pc_rv_i;

// WRITEBACK MULTIPLEXER INTERCONNECTIONS
assign pc_wm_i = pc4_pm_o;
assign mem_wm_i = load_value_mc_o;
assign alu_wm_i = result_a_o;
assign imm_wm_i = immediate_d_o;
assign csr_wm_i = csr_rd_csr_o;
assign is_csr_wm_i = is_csr_d_o;
assign wbsel_wm_i = wbsel_d_o;

// CSR INTERCONNECTIONS
assign en_csr = en_rv_i;
assign clk_csr = clk_rv_i;
assign nrst_csr = nrst_rv_i;
assign is_csr_csr_i = is_csr_d_o;
assign func3_csr_i = func3_d_o;
assign csr_index_csr_i = immediate_d_o;
assign immediate_csr_i = rs1_d_o;
assign rv1_csr_i = rv1_rf_o;
assign external_interrupt_csr_i = external_interrupt_rv_i;
assign illegal_instr_csr_i = illegal_instr_d_o;
assign system_instr_csr_i = system_instr_d_o;
assign instr_misaligned_csr_i = instr_misaligned_f_o;
assign misaligned_load_csr_i = not_aligned_load_mc_o;
assign misaligned_store_csr_i = not_aligned_store_mc_o;
assign pc_csr_i = pc_pm_o;
assign is_mret_csr_i = is_mret_d_o;
```

```
    // INTERRUPT MULTIPLEXER INTERCONNECTIONS
    assign cur_pc_im_i = pc_pm_o;
    assign pc_trap_im_i = trap_address_csr_o;
    assign pc_prev_im_i = return_address_csr_o;
    assign enter_trap_im_i = enter_trap_csr_o;
    assign exit_trap_im_i = exit_trap_csr_o;
    assign sel_out_pc_im_i = wr_pc_rv_i;

    // MODULE TOP-LEVEL PORT INTERCONNECTIONS
    assign reg_value_rv_o = rvdeb_rf_o;
    assign is_ret_rv_o = is_ret_d_o;
    assign pc_rv_o = pc_im_o[ADDR_WIDTH-1:0];
    assign cren_rv_o = read_en_f_o;
    assign dout_rv_o = store_value_mc_o;
    assign addr_rv_o = aligned_addr_mc_o;
    assign mem_type_rv_o = mem_type_mc_o;
    assign mem_mask_rv_o = mem_mask_mc_o;
    assign drwren_rv_o = wr_en_mc_o;
    assign dren_rv_o = rd_en_mc_o;

endmodule
```

code_ram.v

```
module code_ram#(parameter ADDR_WIDTH = 7)(
  input [ADDR_WIDTH-1:0] addra,  // Address bus
  input clk,                     // System clock signal
  input ena,                     // Synchronous read enable signal
  input nrsta,                   // Asynchronous active-low output register reset (does not clear
     memory array cells)
  input wr_en,                   // Synchronous write enable signal
  input [31:0] dina,             // Input data bus
  output [31:0] douta            // RAM output data bus
);
  localparam nrow = 2**ADDR_WIDTH/4;
  reg [31:0] BRAM [nrow-1:0];
  reg [31:0] ram_data;
  wire [ADDR_WIDTH-3:0] address = addra[ADDR_WIDTH-1:2];
  wire unconnected_wires = addra[1:0];

  integer i;
  // Synchronous memory array access and asynchronous output register reset block
  always @(posedge clk, negedge nrsta) begin
    if (!nrsta) begin
      ram_data <= 32'b0;
      for (i = 0; i < nrow; i = i+1) begin
          BRAM[i] <= 32'b0;
      end
    end
    else begin
      if (ena) begin
        ram_data <= BRAM[address];
      end
      else if (wr_en) begin
        BRAM[address] <= dina;
      end
    end
  end

  // Continuous assignment for output data bus routing
  assign douta = ram_data;

endmodule
```

debug_module.v

```
module debug_module
#(  parameter ADDR_WIDTH = 7)(
    // Top-Level Module Interface Signals
    input wire en_i,                          // Global module enable
    input wire clk_i,                         // System clock signal
    input wire nrst_i,                        // Asynchronous active-low reset signal
    input wire is_ret_core_i,                 // Instruction return tracking signal asserted by the core
    input wire core_en_i,                     // Core execution enable control input
    input wire debug_mode_i,                  // Host debugger access mode select (0: Write, 1: Read)
```

```
    input wire [ADDR_WIDTH-1:0] address_i,   // Target address input sampled from external pins
    input wire [1:0] mem_type_i,             // Target memory space descriptor ('11': I/O space, '10':
     Register File, '01': Code RAM, '00': Data RAM)
    input wire [31:0] data_from_out_i,        // Input data payload sampled from external pins for
     write transfers
    output reg [31:0] data_to_out_o = 32'b0,  // Output data payload read out from target internal
     memory or Register File
    output reg data_ready_o = 1'b0,           // Read validation flag; asserted when valid data is
     stable on output pins
    // Internal Core and Memory Control Interface Signals
    output reg [31:0] data_to_int_o = 32'b0,         // Forwarded data payload routed to internal
     system memory blocks
    output reg [ADDR_WIDTH-1:0] address_int_o = 0,   // Forwarded address bus routed from host debugger
      to internal system memory blocks
    output reg [1:0] mem_type_int_o = 2'b0,          // Forwarded memory space descriptor routed to
     internal system memory blocks
    output reg core_en_o = 1'b0,                     // Intercepted execution enable control line
     routed to the Core
    // Code RAM Target Interface
    output reg sel_cr_o = 1'b0,           // Chip-select control line routed to the Code RAM
    output reg wren_cr_o = 1'b0,          // Synchronous write enable control line routed to the Code
     RAM
    input wire [31:0] data_from_cr_i,     // Input data payload read out from the Code RAM
    // Data RAM Target Interface
    output reg sel_dr_o = 1'b0,           // Chip-select control line routed to the Data RAM
    output reg wren_dr_o = 1'b0,          // Synchronous write enable control line routed to the Data
     RAM
    input wire [31:0] data_from_dr_i,     // Input data payload read out from the Data RAM
    // Register File Target Interface
    output reg wren_reg_o = 1'b0,         // Synchronous write enable control line routed to the
     Register File
    input wire [31:0] data_from_reg_i,    // Input data payload read out from the Register File
    // Program Counter (PC) Control Interface
    output reg wr_pc_o = 1'b0,
    output reg [31:0] out_pc_o = 32'b0
    );

     // Host Debugger Protocol Operation Modes
     /* DEBUG HOST OPERATION SPECIFICATION:
        00 (DEB_READ):  Execution core enters stall state; targeted data is read from internal memory/
     registers based on address bus configuration.
        01 (DEB_WRITE): Execution core enters stall state; targeted data is committed to internal
     memory/registers based on address bus configuration. */
     localparam DEB_READ = 0;
     localparam DEB_WRITE = 1;
     localparam PC_ADDR = {ADDR_WIDTH{1'b0}};

     /* DEFAULT HARDWARE MEMORY SPACE MAP ASSIGNMENTS:
        From address '00 000 0000' to '00 111 1100' -> TARGET: DATA RAM
        From address '01 000 0000' to '01 111 1100' -> TARGET: CODE RAM
        From address '10 000 0000' to '10 001 1111' -> TARGET: REGISTER FILE */

     reg [31:0] data_to_out;
     reg [31:0] data_to_int;
     reg [ADDR_WIDTH-1:0] address_int;
     reg [1:0] mem_type_int;
     reg [1:0] mem_type_int_1;
     reg sel_cr;
     reg wren_cr;
     reg sel_dr;
     reg wren_dr;
     reg wren_reg;
     reg core_en;
     reg wr_pc;
     reg data_ready;
     reg data_ready_1;
     reg data_ready_2;
     // Host Debugger Configuration and Hardware Control Registers
     localparam INT_ON_RET = {2'b11, {(ADDR_WIDTH-1){1'b0}},1'b1}; // Memory-mapped address tracking
     configuration register 'creg_int_on_ret'
     reg creg_int_on_ret;

     // Synchronous Finite State and Datapath Routing Process Block
     always @(posedge clk_i, negedge nrst_i) begin
          if (!nrst_i) begin
               data_to_out <= 32'b0;
               data_to_int <= 32'b0;
               address_int <= {(ADDR_WIDTH){1'b0}};
               mem_type_int <= 2'b0;
               mem_type_int_1 <= 2'b0;
               sel_cr <= 1'b0;
               wren_cr <= 1'b0;
               sel_dr <= 1'b0;
               wren_dr <= 1'b0;
               wren_reg <= 1'b0;
               wr_pc <= 1'b1;
               core_en <= 1'b0;
```

```
                data_ready <= 1'b0;
                data_ready_1 <= 1'b0;
                data_ready_2 <= 1'b0;
                data_ready_o <= 1'b0;
                creg_int_on_ret <= 1'b1;
                data_to_out_o <= 32'b0;
                data_to_int_o <= 32'b0;
                address_int_o <= 0;
                mem_type_int_o <= 2'b0;
                core_en_o <= 1'b0;
                sel_cr_o <= 1'b0;
                wren_cr_o <= 1'b0;
                sel_dr_o <= 1'b0;
                wren_dr_o <= 1'b0;
                wren_reg_o <= 1'b0;
                wr_pc_o <= 1'b1;
                out_pc_o <= 32'b0;
           end
           else begin
                if (en_i) begin
                     data_to_int_o <= data_to_int;
                     address_int_o <= address_int;
                     mem_type_int_o <= mem_type_int;
                     mem_type_int_1 <= mem_type_int_o;
                     sel_cr_o <= sel_cr;
                     wren_cr_o <= wren_cr;
                     sel_dr_o <= sel_dr;
                     wren_dr_o <= wren_dr;
                     wren_reg_o <= wren_reg;
                     core_en_o <= core_en;
                     wr_pc_o <= wr_pc;
                     if (is_ret_core_i && creg_int_on_ret) begin
                          core_en_o <= 1'b0;
                          core_en <= 1'b0;
                     end
                     data_ready_1 <= data_ready;
                     data_ready_2 <= data_ready_1;
                     data_ready_o <= data_ready_2;
                     if (data_ready_1) begin
                          case (mem_type_int_o)
                               `DR_MEM:  data_to_out <= data_from_dr_i;
                               `REG_MEM: data_to_out <= data_from_reg_i;
                               default:  data_to_out <= 32'b0;
                          endcase
                     end
                     if (data_ready_2) begin
                          case (mem_type_int_1)
                               `CR_MEM:  data_to_out_o <= data_from_cr_i;
                               `DR_MEM:  data_to_out_o <= data_to_out;
                               `REG_MEM: data_to_out_o <= data_to_out;
                               default:  data_to_out <= 32'b0;
                          endcase
                     end
                     case (debug_mode_i)
                          DEB_READ:
                               begin
                                    data_ready <= 1'b1;
                                    data_to_int <= 32'b0;
                                    address_int <= address_i;
                                    mem_type_int <= mem_type_i;
                                    sel_cr <= (mem_type_i == `CR_MEM);
                                    wren_cr <= 1'b0;
                                    sel_dr <= (mem_type_i == `DR_MEM);
                                    wren_dr <= 1'b0;
                                    wren_reg <= 1'b0;
                                    core_en <= 1'b0;
                               end
                          DEB_WRITE:
                               begin
                                    data_ready <= 1'b0;
                                    data_to_int <= data_from_out_i;
                                    address_int <= address_i;
                                    mem_type_int <= mem_type_i;
                                    sel_cr <= (mem_type_i == `CR_MEM);
                                    wren_cr <= (mem_type_i == `CR_MEM);
                                    sel_dr <= (mem_type_i == `DR_MEM);
                                    wren_dr <= (mem_type_i == `DR_MEM);
                                    wren_reg <= (mem_type_i == `REG_MEM);
                                    if ((address_i == PC_ADDR) && (mem_type_i == `IO_MEM)) begin
                                         wr_pc <= 1'b1;
                                         out_pc_o <= data_from_out_i;
                                    end
                                    core_en <= 1'b0;
                                    if (address_i == INT_ON_RET)
                                         creg_int_on_ret <= data_to_int[0];
                               end
                     endcase
```

```
                end
                else if (!en_i) begin
                     data_ready <= 1'b0;
                     data_to_int_o <= data_to_int;
                     address_int_o <= address_int;
                     mem_type_int_o <= mem_type_int;
                     mem_type_int_1 <= mem_type_int_o;
                     sel_cr_o <= sel_cr;
                     wren_cr_o <= wren_cr;
                     sel_dr_o <= sel_dr;
                     wren_dr_o <= wren_dr;
                     wren_reg_o <= wren_reg;
                     core_en_o <= core_en;
                     wr_pc_o <= wr_pc;
                     data_ready_1 <= data_ready;
                     data_ready_2 <= data_ready_1;
                     data_ready_o <= data_ready_2;
                     if (data_ready_1) begin
                          case (mem_type_int_o)
                               `DR_MEM:  data_to_out <= data_from_dr_i;
                               `REG_MEM: data_to_out <= data_from_reg_i;
                               default:  data_to_out <= 32'b0;
                          endcase
                     end
                     if (data_ready_2) begin
                          case (mem_type_int_1)
                               `CR_MEM:  data_to_out_o <= data_from_cr_i;
                               `DR_MEM:  data_to_out_o <= data_to_out;
                               `REG_MEM: data_to_out_o <= data_to_out;
                               default:  data_to_out_o <= 32'b0;
                          endcase
                     end
                     if (core_en) begin
                          wr_pc <= 1'b0;
                     end
                     if (is_ret_core_i && creg_int_on_ret) begin
                          core_en_o <= 1'b0;
                          core_en <= 1'b0;
                     end
                     else begin
                          core_en <= core_en_i;
                     end
                     data_to_int <= 32'b0;
                     address_int <= 0;
                     mem_type_int <= 2'b0;
                     sel_cr <= 1'b0;
                     wren_cr <= 1'b0;
                     sel_dr <= 1'b0;
                     wren_dr <= 1'b0;
                     wren_reg <= 1'b0;
                     data_ready <= 1'b0;
                end
           end
      end

endmodule
```

data_ram_mux.v

```
`define DR_MEM  2'b00
`define CR_MEM  2'b01
`define REG_MEM  2'b10
`define IO_MEM  2'b11

module data_ram_mux#( parameter ADDR_WIDTH = 7)(
        // RISC-V CORE INTERFACE SIGNALS
    input wire [ADDR_WIDTH-1:0] addr_rv_i,  // Target routing address for core memory transactions
    input wire [1:0] mem_type_rv_i,         // Memory space classification bits (00: Data space, 11:
     Memory-mapped I/O)
    input wire [31:0] dout_rv_i,            // Data payload forwarded from the core to the Data RAM or
     Peripheral subsystems
    input wire [3:0] mem_mask_rv_i,         // Byte-enable mask from the core processing unit
    input wire drwren_rv_i,                 // Synchronous write enable control line from the core
        input wire en_rv_i,                     // Synchronous read enable control line from the core
    // INPUT SUBSYSTEM FEEDBACK SIGNALS
    input wire [31:0] data_from_dr_i,       // Data payload read from the Data RAM block
    input wire [31:0] data_from_gpo_i,      // Data payload read from the Peripheral General Purpose
     Output controller
    // OUTPUT PORT ROUTED TO THE CORE
    output wire [31:0] data_to_core_o,      // Multiplexed read data bus routed back to the processor
     core
```

```
        // HOST DEBUGGER MODULE INTERFACE SIGNALS
    input wire [ADDR_WIDTH-1:0] addr_deb_i, // Target routing address for host debugger transactions
    input wire [1:0] mem_type_deb_i,        // Memory space classification bits from host debugger (00:
      Data space, 11: Memory-mapped I/O)
    input wire [31:0] dout_deb_i,           // Data payload forwarded from the debugger to internal
     memory or Peripheral subsystems
    // Note: The memory byte mask is implicitly driven to 4'b1111 for all write operations issued by
     the DEBUG module
    input wire drwren_deb_i,                // Synchronous write enable control line from host debugger
    // Note: The debug read enable control line is implicitly decoded from the state of 'sel_deb_i' and
      'drwren_deb_i'
    input wire sel_deb_i,                   // Multiplexer control select line (0: RISC-V Core mode, 1:
      Host Debugger mode)
    // TARGET INTERFACE OUTPUT PORTS
    output wire [ADDR_WIDTH-1:0] addr_o,  // Multiplexed target memory address bus
    output wire [31:0] dout_o,            // Multiplexed write data payload bus
    output wire [3:0] mem_mask_o,         // Multiplexed peripheral/memory byte mask
    output wire out_wren_o,               // Synchronous write enable control line routed to Peripheral
      I/O space
    output wire out_en_o,                 // Synchronous read enable control line routed to Peripheral
     I/O space
    output wire drwren_o,                 // Synchronous write enable control line routed to the Data
     RAM
    output wire dren_o                    // Synchronous read enable control line routed to the Data
     RAM
    );
    wire wren;
    wire en;
    // Multiplexer routing logic for basic datapath configurations
    assign addr_o = (sel_deb_i == 1'b0)? addr_rv_i : addr_deb_i;
    assign dout_o = (sel_deb_i == 1'b0)? dout_rv_i : dout_deb_i;
    assign mem_mask_o = (sel_deb_i == 1'b0)? mem_mask_rv_i : 4'b1111;
    // Internal decoded transaction control attributes
    wire [1:0] mem_type = (sel_deb_i == 1'b0)? mem_type_rv_i : mem_type_deb_i;
    assign wren = (sel_deb_i == 1'b0)? drwren_rv_i : drwren_deb_i;
    assign en = (sel_deb_i == 1'b0)? en_rv_i : !(drwren_deb_i);
    // Decoding and gating logic for Data RAM target transactions
    assign drwren_o = wren && (mem_type == `DR_MEM);
    assign dren_o = en && (mem_type == `DR_MEM);
    // Decoding and gating logic for Memory-Mapped I/O target transactions
    assign out_wren_o = wren && (mem_type == `IO_MEM);
    assign out_en_o = en && (mem_type == `IO_MEM);
    // Read datapath multiplexing logic routed to the core interface
    assign data_to_core_o = (en && (mem_type == `IO_MEM))? data_from_gpo_i : data_from_dr_i;

endmodule
```

code_ram_mux.v

```
module code_ram_mux#( parameter ADDR_WIDTH = 7 )(
    // RISC-V CORE INTERFACE SIGNALS
    input wire [ADDR_WIDTH-1:0] addr_rv_i,     // Program Counter (PC) generated by the core unit
    input wire en_rv_i,                        // Instruction fetch read enable control line from the
     core
    // HOST DEBUGGER MODULE INTERFACE SIGNALS
    input wire [ADDR_WIDTH-1:0] addr_deb_i,  // Target routing address for host debugger code
     transactions
    input wire wren_deb_i,                   // Synchronous write enable control line from host
     debugger
    input wire [31:0] data_from_deb,         // Instruction payload forwarded from host debugger for
     write transfers
    input wire sel_cr_deb_i,                 // Multiplexer control select line (0: RISC-V Core mode,
     1: Host Debugger mode)
    // CODE RAM TARGET INTERFACE PORTS
    output wire [ADDR_WIDTH-1:0] addr_o,     // Multiplexed target memory address bus
    output wire en_o,                        // Multiplexed synchronous read enable control line
    output wire wren_o,                      // Multiplexed synchronous write enable control line
    output wire [31:0] data_to_cr_o          // Multiplexed instruction payload bus
    );

    // Multiplexer routing logic for basic datapath configurations
    assign addr_o = (sel_cr_deb_i == 1'b0)? addr_rv_i : addr_deb_i;
    assign en_o = (sel_cr_deb_i == 1'b0)? en_rv_i : !(wren_deb_i);
    assign wren_o = (sel_cr_deb_i == 1'b0)? 1'b0 : wren_deb_i;
    assign data_to_cr_o = (sel_cr_deb_i == 1'b0)? 32'b0 : data_from_deb;

endmodule
```

gpo_controller.v

```
module gpo_controller#( parameter ADDR_WIDTH = 7,
    parameter GPO_NUM = 8,
    parameter GPO_ADDR_WIDTH = 2)(
    input wire clk,
    input wire nrst,
    input wire en_i,                          // Peripheral controller enable control line from the
     core
    input wire [ADDR_WIDTH-1:0] address_i,    // Memory-mapped I/O target address from the ALU unit
    input wire wr_en_i,
    input wire [3:0] mem_mask_i,              // Byte-enable memory mask for selective load/store
     access (Word, Halfword, Byte)
    input wire [31:0] data_from_core_i,       // Write data payload forwarded from the processor
     Register File
    input wire we_from_out,
    output wire [31:0] data_to_core_o,        // Multiplexed read data payload routed back to the
     processor Register File
    output wire [GPO_NUM-1:0] data_to_out_o,
    input wire [31:0] data_from_serial_i,
    output wire [31:0] data_to_serial_o
    );

    reg [7:0] BRAM [15:4];
    reg [7:0] ADC_REG[3:0];
    wire [GPO_ADDR_WIDTH-1:0] address = address_i[GPO_ADDR_WIDTH+1:2];
    wire unconnected_wires = address_i[1:0];
    wire wea = (address != 0) && (wr_en_i && (address_i[ADDR_WIDTH-1:GPO_ADDR_WIDTH+2] == {(ADDR_WIDTH-
     GPO_ADDR_WIDTH-2){1'b1}}));
    wire weadc = (wr_en_i && (address == 0));
    integer i;
    // Synchronous write and external overwrite process block for the Analog-to-Digital Converter (ADC)
      registers
    always @(posedge clk, negedge nrst) begin
        if (!nrst) begin
            for (i = 0; i < 4; i = i+1) begin
                ADC_REG[i] <= 8'b0;
            end
        end
        else if (we_from_out) begin
            ADC_REG[0] <= data_from_serial_i[7:0];
            ADC_REG[1] <= data_from_serial_i[15:8];
            ADC_REG[2] <= data_from_serial_i[23:16];
            ADC_REG[3] <= data_from_serial_i[31:24];
        end
        else if (weadc) begin
            if(mem_mask_i[0])
                ADC_REG[0] <= data_from_core_i[7:0];
            if(mem_mask_i[1])
                ADC_REG[1] <= data_from_core_i[15:8];
            if(mem_mask_i[2])
                ADC_REG[2] <= data_from_core_i[23:16];
            if(mem_mask_i[3])
                ADC_REG[3] <= data_from_core_i[31:24];
        end
    end

    // Synchronous byte-masked write process block for the scratchpad memory array (BRAM)
    always @(posedge clk, negedge nrst) begin
        if (!nrst) begin
            for (i = 4; i < 16; i = i+1) begin
                BRAM[i] <= 8'b0;
            end
        end
        else if (wea) begin
            if(mem_mask_i[0])
                BRAM[{address,2'b00}] <= data_from_core_i[7:0];
            if(mem_mask_i[1])
                BRAM[{address,2'b01}] <= data_from_core_i[15:8];
            if(mem_mask_i[2])
                BRAM[{address,2'b10}] <= data_from_core_i[23:16];
            if(mem_mask_i[3])
                BRAM[{address,2'b11}] <= data_from_core_i[31:24];
        end
    end

    // Word reconstruction logic from byte-addressed memory structures
    wire [31:0] bram_reg = {BRAM[{address,2'b11}],BRAM[{address,2'b10}],BRAM[{address,2'b01}],BRAM[{
     address,2'b00}]};
    wire [31:0] adc_reg = {ADC_REG[{address,2'b11}],ADC_REG[{address,2'b10}],ADC_REG[{address,2'b01}],
     ADC_REG[{address,2'b00}]};
    // Peripheral address decoding and read multiplexing stages
    wire [31:0] data_to_core = (address == 0)?  adc_reg : bram_reg;
    wire [31:0] data = {BRAM[15],BRAM[14],BRAM[13],BRAM[12]};
```

```
    // Continuous assignment for serial interface, general purpose outputs, and core read bus routing
    assign data_to_serial_o = {BRAM[11],BRAM[10],BRAM[9],BRAM[8]};
    assign data_to_out_o = data[GPO_NUM-1:0];
    assign data_to_core_o = (en_i == 1'b0)? 32'b0 : data_to_core;

endmodule
```

fetch.v

```
module fetch(
    input wire en_i,               // Global module enable
    input wire clk_i,              // System clock signal
    input wire nrst_i,             // Asynchronous active-low reset signal for the Program Counter (PC)
    input wire [31:0] next_pc_i,   // Next Program Counter input target
    output wire read_en_o,         // Synchronous read enable control line routed to the Code RAM
    output reg [31:0] pc_o,        // Current Program Counter register output
    output wire instr_misaligned_o // Instruction address misaligned exception flag
    );


    // Continuous assignment for read enable gating and exception detection logic
    assign read_en_o = nrst_i && en_i;
    assign instr_misaligned_o = pc_o[1:0] != 2'b0;
    // Synchronous Program Counter register update process block
    always @(posedge clk_i, negedge nrst_i)
    begin
        if (!nrst_i) begin
            pc_o <= 32'b0;
        end
        else begin
            if (en_i) begin
                pc_o <= next_pc_i;
            end
        end
    end

endmodule
```

alu_mux.v

```
module alu_mux(
    input wire [31:0] pc_i,      // Current Program Counter (PC) value
    input wire [31:0] rv1_i,     // Operand value read from Register File source 1 (rs1)
    input wire v1_sel_i,         // Routing selection line generated by the Instruction Decoder stage
    input wire [31:0] rv2_i,     // Operand value read from Register File source 2 (rs2)
    input wire [31:0] imm_i,     // Extended immediate value vector formatted by the Instruction
     Decoder stage
    input wire v2_sel_i,         // Routing selection line generated by the Instruction Decoder stage
    output wire [31:0] value1_o, // Multiplexed primary execution operand forwarded to ALU input 1 (PC
     vs. RV1)
    output wire [31:0] value2_o  // Multiplexed secondary execution operand forwarded to ALU input 2 (
     IMM vs. RV2)
    );

    // Continuous assignment mapping for operand processing datapaths
    assign value1_o = (v1_sel_i == `V1_PC)? pc_i : rv1_i;
    assign value2_o = (v2_sel_i == `V2_IMM)? imm_i : rv2_i;

endmodule
```

writeback.v

```
module writeback(
    input wire [31:0] pc_i,        // Incremente Program Counter value (PC + 4) forwarded from the PC
     selection stage
    input wire [31:0] mem_i,       // Data payload loaded from the target memory space
    input wire [31:0] alu_i,       // Execution result forwarded from the ALU subsystem
    input wire [31:0] imm_i,       // Extended immediate vector formatted by the Instruction Decoder
     stage
```

```
    input wire [31:0] csr_i,       // Register value read from the Control and Status Registers (CSR)
     block
    input wire is_csr_i,           // Control signal asserted for Control and Status Register
     instructions
    input wire [1:0] wbsel_i,      // Multiplexer routing selection code from the Instruction Decoder
     stage
    output wire [31:0] rd_value_o  // Multiplexed writeback data payload routed to the Register File
     destination register (rd)
    );

    // Continuous assignment mapping for the writeback datapath selection stages
    assign rd_value_o = is_csr_i?               csr_i :
                        (wbsel_i == `WB_PC)?     pc_i :
                        (wbsel_i == `WB_MEM)?    mem_i :
                        (wbsel_i == `WB_ALU)?    alu_i :
                                                 imm_i ;
endmodule
```

register_file.v

```
module register_file(
    input wire en,                   // Control enable line used to stall the Register File stage
    input wire clk,
    input wire nrst,                 // Asynchronous active-low reset signal
    input wire [4:0] rs1_i,          // Target routing address for source register 1 (rs1)
    input wire [4:0] rs2_i,          // Target routing address for source register 2 (rs2)
    input wire [4:0] rd_i,           // Target routing address for the destination register (rd)
    input wire wr_en_i,              // Synchronous write enable control line from the core
    input wire [31:0] rd_value_i,    // Data payload to be committed into the destination register (rd)
    input wire wr_en_deb_i,          // Synchronous write enable control line from the external host
     debugger module
    input wire [4:0] rsdeb_i,        // Target register address designated for debug operations
    input wire [31:0] rwr_deb_i,     // Data payload to be committed into the register specified by
     rsdeb_i
    output wire [31:0] rvdeb_o,      // Data value read from the Register File and routed to the Debug
     module
    output wire [31:0] rv1_o,        // Output data operand read from the source register 1 (rs1)
    output wire [31:0] rv2_o         // Output data operand read from the source register 2 (rs2)
    );
    reg [31:0] local_reg [31:1];     // Internal hardware memory array implementing RISC-V registers x1
      to x31
    wire we;
    wire wr_en_deb;
    // Gating logic to enforce the hardwired zero constraint on register x0
    assign we = (rd_i==1'b0)? 1'b0 : wr_en_i;
    assign wr_en_deb = (rsdeb_i==1'b0)? 1'b0 : wr_en_deb_i;
        // ASYNCHRONOUS READ OPERATIONS ////////////////////////////////
        // Implementation of the structural hardwired zero for x0 during operand fetching
        assign rv1_o = rs1_i == 0? 32'b0 : local_reg[rs1_i];
        assign rv2_o = rs2_i == 0? 32'b0 : local_reg[rs2_i];
        assign rvdeb_o = rsdeb_i == 0? 32'b0 : local_reg[rsdeb_i];
        // SYNCHRONOUS WRITE OPERATIONS ////////////////////////////////
        integer i;
        always @(posedge clk or negedge nrst)
        begin
                if(!nrst)
                begin
                        for(i = 1; i <= 31; i = i + 1)
                                local_reg[i] <= 32'b0;
                end
                else begin
                        if(we && en)
                                local_reg[rd_i] <= rd_value_i;
                    else if (wr_en_deb)
                        local_reg[rsdeb_i] <= rwr_deb_i;
                end
        end

endmodule
```

pc_sel.v

```
module pc_sel(
    input [31:0] pc_i,       // Current Program Counter (PC) register output from the Fetch stage
    input [31:0] pc_alu_i,   // Target execution address calculated by the ALU subsystem
    input [31:0] out_pc_i,   // Target address vector forced from an external interface pin
    input is_jump_i,         // Control signal asserted for unconditional jump instructions
    input branch_result_i,   // Evaluation result generated by the conditional branch comparison logic
    input sel_pc_i,          // Multiplexer control select line (forces the output to the external
     target address if asserted)
    output [31:0] pc_o,      // Multiplexed next Program Counter (PC) target bus routed to the Fetch
     stage
    output [31:0] pc4_o      // Incremented Program Counter value (PC + 4) destined for the writeback
     datapath
    );
    // Continuous assignment for sequential address generation
    assign pc4_o = pc_i+4;
    // Multiplexer routing logic for next Program Counter (PC) target resolution
    assign pc_o = (sel_pc_i)?          out_pc_i :
    (is_jump_i || branch_result_i)?    pc_alu_i :
                                       pc4_o ;

endmodule
```

memory_controller.v

```
module memory_controller#(  parameter ADDR_WIDTH = 7)(
    input wire en_i,                   // Control enable line used to stall memory controller
     transactions from the core
    input wire [31:0] address_i,       // Target execution address forwarded from the ALU unit
    input wire [1:0] mem_size_i,       // Memory transaction data width descriptor (Word, Halfword,
     Byte)
    input wire uload_i,                // Control signal asserted for zero-extended (unsigned) load
     instructions
    input wire [31:0] store_value_i,   // Write data payload forwarded from the processor Register File
    input wire [31:0] load_value_i,    // Unformatted raw data payload read from the Data Memory
    input wire is_load_i,              // Control signal asserted for load operations
    input wire is_store_i,             // Control signal asserted for store operations
    output reg [31:0] store_value_o,   // Formatted write data payload routed to the Data Memory
    output wire [31:0] load_value_o,   // Formatted and sign/zero-extended read data payload routed to
     the writeback stage
    output wire wr_en_o,               // Synchronous write enable control line routed to the Data
     Memory
    output wire rd_en_o,               // Synchronous read enable control line routed to the Data
     Memory
    output reg [3:0] mem_mask_o,       // Active-byte write enable mask destined for the Data Memory
    output wire not_aligned_load_o,    // Exception flag asserted upon an unaligned address condition
     during a load instruction
    output wire not_aligned_store_o,   // Exception flag asserted upon an unaligned address condition
     during a store instruction
    output wire [ADDR_WIDTH-1:0] aligned_addr_o, // Base word-aligned address generated by the memory
     controller
    output wire [1:0] mem_type_o
    );
    // Continuous assignment for base word alignment and memory space decoding
    assign aligned_addr_o = {address_i[ADDR_WIDTH-1:2],2'b00};
    assign mem_type_o = {address_i[ADDR_WIDTH+1:ADDR_WIDTH]};
    // Bounds check to detect access violation beyond the physical memory boundary
    wire out_of_memory = (address_i[31:ADDR_WIDTH+2] != 0);
    // Combinational evaluation process block for unaligned address exception detection
    reg not_aligned;
    always @(mem_size_i or address_i or is_store_i or is_load_i) begin
        case (mem_size_i)
            `MEM_B :   not_aligned <= 1'b0;
            `MEM_H :   not_aligned <= address_i[0];
            `MEM_W :   not_aligned <= (address_i[1:0] != 2'b00);
            default:   not_aligned <= (is_store_i | is_load_i);
        endcase
    end

    // Gating logic to isolate unaligned exceptions based on instruction type
    assign not_aligned_load_o = (not_aligned && is_load_i);
    assign not_aligned_store_o = (not_aligned && is_store_i);

    // Combinational evaluation process block for byte-enable write mask generation
    always @(mem_size_i or address_i) begin
        case (mem_size_i)
            `MEM_B :   mem_mask_o <= 4'b1 << address_i[1:0];
```

```
            `MEM_H :   mem_mask_o <= 4'b11 << address_i[1:0];
            `MEM_W :   mem_mask_o <= 4'b1111 << address_i[1:0];
            default:   mem_mask_o <= 4'b0;
        endcase
    end

    // Combinational evaluation process block for write data formatting and byte steering
    always @(mem_size_i or address_i or store_value_i) begin
        case (mem_size_i)
            `MEM_B :   store_value_o <= store_value_i[7:0] << address_i[1:0]*8;
            `MEM_H :   store_value_o <= store_value_i[15:0] << address_i[1:0]*8;
            `MEM_W :   store_value_o <= store_value_i[31:0] << address_i[1:0]*8;
            default:   store_value_o <= 32'b0;
        endcase
    end

    // Byte-level conditional extraction and sign/zero-extension datapath logic
    wire [31:0] load_b_00 = {{24{((!uload_i) & (load_value_i[7]))}},load_value_i[7:0]};
    wire [31:0] load_b_01 = {{24{((!uload_i) & (load_value_i[15]))}},load_value_i[15:8]};
    wire [31:0] load_b_10 = {{24{((!uload_i) & (load_value_i[23]))}},load_value_i[23:16]};
    wire [31:0] load_b_11 = {{24{((!uload_i) & (load_value_i[31]))}},load_value_i[31:24]};

    // Halfword-level conditional extraction and sign/zero-extension datapath logic
    wire [31:0] load_h_00 = {{16{((!uload_i) & (load_value_i[15]))}},load_value_i[15:0]};
    wire [31:0] load_h_01 = {{16{((!uload_i) & (load_value_i[23]))}},load_value_i[23:8]};
    wire [31:0] load_h_10 = {{16{((!uload_i) & (load_value_i[31]))}},load_value_i[31:16]};
    wire [31:0] load_h_11 = {{24{((!uload_i) & (load_value_i[31]))}},load_value_i[31:24]};

    // Routing multiplexer select signal generation for transaction sizing
    wire [1:0] load_sel = ((!is_load_i) | mem_size_i);

    // Byte and halfword steering multiplexer configurations
    wire [31:0] load_b = address_i[1] ? (address_i[0] ? load_b_11 : load_b_10) : (address_i[0] ?
     load_b_01 : load_b_00);
    wire [31:0] load_h = address_i[1] ? (address_i[0] ? load_h_11 : load_h_10) : (address_i[0] ?
     load_h_01 : load_h_00);
    wire [31:0] load_w = load_value_i>>8*(address_i[1:0]);

    // Continuous assignment for readback datapath routing and memory enable control lines
    assign load_value_o = load_sel[1] ? (load_sel[0] ? 32'b0 : load_w) : (load_sel[0] ? load_h : load_b
     );
    assign wr_en_o = is_store_i && en_i && !out_of_memory;
    assign rd_en_o = is_load_i && en_i && !out_of_memory;

endmodule
```

interrupt_mux.v

```
module interrupt_mux(
    input wire [31:0] cur_pc_i,   // Current Program Counter (PC) target forwarded from the PC
     selection stage
    input wire [31:0] pc_trap_i,  // Trap vector address generated by the CSR exception sub-module
    input wire [31:0] pc_prev_i,  // Saved return address target (mepc) to be restored upon trap
     handling completion
    input wire enter_trap_i,      // Control signal asserted prior to entering an exception trap state
    input wire exit_trap_i,       // Control signal asserted prior to executing a trap return
     instruction (mret)
    input wire sel_out_pc_i,      // Multiplexer override selection flag (forces the PC from an
     external source if asserted)
    output wire [31:0] pc_o       // Multiplexed target address bus routed to the main Fetch stage
     pipeline loop
    );
    // Gating logic to prevent trap routine vectoring during external debugger overrides
    wire enter_trap = ((enter_trap_i) && (!sel_out_pc_i));
    wire exit_trap = ((exit_trap_i) && (!sel_out_pc_i));

    // Multiplexer routing logic for exception vectoring and execution context restoration
    assign pc_o = (enter_trap)?   pc_trap_i :
                  (exit_trap)?    pc_prev_i :
                                  cur_pc_i ;

endmodule
```

decode.v

```
module decode(
    input wire [31:0] instr_i,      // Fetched instruction word received from the Fetch stage
    output wire [4:0] rd_o,         // Target routing address for the destination register (rd)
    output wire [4:0] rs1_o,        // Target routing address for source register 1 (rs1)
    output wire [4:0] rs2_o,        // Target routing address for source register 2 (rs2)
    output wire [2:0] func3_o,      // func3 field extracted from the instruction word
    output reg [31:0] immediate_o,  // Decoded and sign-extended immediate value vector
    output wire is_csr_o,           // Control signal asserted for Control and Status Register (CSR)
     instructions
    output wire rd_we_o,            // Synchronous write enable control line routed to the Register
     File
    output reg [4:0] alu_op_o,      // 5-bit control opcode defining the specific ALU operation
    output wire v1sel_o,            // Multiplexer control select line for ALU operand 1 (RV1 vs. PC)
    output wire v2sel_o,            // Multiplexer control select line for ALU operand 2 (RV2 vs. IMM)
    output wire [1:0] wbsel_o,      // Multiplexer control select line defining the writeback data
     payload source
    output wire is_load_o,          // Synchronous read enable control line routed to the Data Memory
    output wire is_store_o,         // Synchronous write enable control line routed to the Data Memory
    output wire [1:0] mem_size_o,   // Memory transaction data width descriptor (Byte, Halfword, Word)
    output wire uload_o,            // Control signal asserted for zero-extended (unsigned) load
     instructions
    output wire is_jalr_o,          // Control signal asserted when the decoded instruction is a JALR
    output wire is_jump_o,          // Control signal asserted for unconditional jump instruction
     formats (J-type)
    output wire is_mret_o,          // Control signal asserted when the decoded instruction is an MRET
     (Trap Return)
    output wire is_ret_o,           // Control signal asserted when the decoded instruction matches the
      RET pseudo-op
    output wire illegal_instr_o,    // Exception flag asserted upon detection of an illegal or
     undefined instruction
    output wire system_instr_o      // Control signal asserted for environmental system instructions (
     EBREAK or ECALL)
    );

    // Continuous assignment for structural field extraction and macro comparison
    assign rs1_o = instr_i[19:15];
    assign rs2_o = instr_i[24:20];
    assign rd_o = instr_i[11:7];
    assign is_mret_o = (instr_i == `MRET)? 1'b1 : 1'b0;
    assign is_ret_o = (instr_i == `RET)? 1'b1 : 1'b0;
    assign func3_o = instr_i[14:12];
    wire [6:0] opcode = instr_i[6:0];
    wire [6:0] func7 = instr_i[31:25];

    ////////////////////////////////////////////////////////////////////////////////
    // INSTRUCTION FORMAT TYPE DECODING
    ////////////////////////////////////////////////////////////////////////////////
    reg [2:0] ins_type;

    // Combinational evaluation process block for base format classification based on the opcode field
    always @(opcode) begin
        case (opcode)
            `OP_LUI :   ins_type <= `U_TYPE;
            `OP_AUIPC : ins_type <= `U_TYPE;
            `OP_JAL :   ins_type <= `J_TYPE;
            `OP_STORE : ins_type <= `S_TYPE;
            `OP_BRANCH :ins_type <= `B_TYPE;
            `OP_OP :    ins_type <= `R_TYPE;
            `OP_FENCE : ins_type <= `OTHER_TYPE;
            `OP_SYSTEM :ins_type <= `I_TYPE;
            `OP_OPIMM : ins_type <= `I_TYPE;
            `OP_LOAD :  ins_type <= `I_TYPE;
            `OP_JALR :  ins_type <= `I_TYPE;
            default :   ins_type <= `UNDEFINED_TYPE;
        endcase
    end

    ////////////////////////////////////////////////////////////////////////////////
    // IMMEDIATE GENERATION AND FORMAT PATTERNS
    ////////////////////////////////////////////////////////////////////////////////
    wire [31:0] i_immediate = {{20{instr_i[31]}},instr_i[31:20]};
    wire [31:0] s_immediate = {{20{instr_i[31]}}, instr_i[31:25],instr_i[11:7] };
    wire [31:0] b_immediate = {{19{instr_i[31]}}, instr_i[31],instr_i[7],instr_i[30:25],instr_i[11:8],
     1'b0 };
    wire [31:0] u_immediate = {instr_i[31:12], 12'b0 };
    wire [31:0] j_immediate = {{12{instr_i[31]}},instr_i[19:12],instr_i[20],instr_i[30:21],1'b0};

    ////////////////////////////////////////////////////////////////////////////////
    // UNCONDITIONAL CONTROL FLOW (JUMP) DECODING
    ////////////////////////////////////////////////////////////////////////////////
    assign is_jump_o = ((opcode == `OP_JALR) || (opcode == `OP_JAL));
    assign is_jalr_o = (opcode == `OP_JALR);
```

```
////////////////////////////////////////////////////////////////////////////////
// ARITHMETIC / LOGICAL REGISTER-BASED TRANSACTION EVALUATION
////////////////////////////////////////////////////////////////////////////////
wire is_op = ((opcode == `OP_OPIMM) || (opcode == `OP_OP));

////////////////////////////////////////////////////////////////////////////////
// ARITHMETIC LOGIC UNIT (ALU) OPERAND ROUTING SELECTION
////////////////////////////////////////////////////////////////////////////////
wire v1_check = !(opcode == `OP_AUIPC);
wire v2_check = ((opcode == `OP_BRANCH) || (opcode == `OP_OP));
assign v1sel_o = (v1_check)? `V1_RV1 : `V1_PC;
assign v2sel_o = (v2_check)? `V2_RV2 : `V2_IMM;

////////////////////////////////////////////////////////////////////////////////
// CONTROL AND STATUS REGISTER (CSR) LOGIC EVALUATION
////////////////////////////////////////////////////////////////////////////////
assign is_csr_o = ((opcode == `OP_SYSTEM) && (func3_o[1:0] != 2'b0));

////////////////////////////////////////////////////////////////////////////////
// WRITEBACK TARGET SOURCE DECODING
////////////////////////////////////////////////////////////////////////////////
wire wbimm_check = (opcode == `OP_LUI);
wire wbalu_check = ((opcode == `OP_AUIPC) || is_op);
wire wbmem_check = (opcode == `OP_LOAD);
assign wbsel_o = wbimm_check?   `WB_IMM :
                 wbalu_check?   `WB_ALU :
                 wbmem_check?   `WB_MEM :
                                `WB_PC;

////////////////////////////////////////////////////////////////////////////////
// MEMORY SUBSYSTEM ACCESS (LOAD / STORE) EVALUATION
////////////////////////////////////////////////////////////////////////////////
assign is_load_o = (opcode == `OP_LOAD);
assign is_store_o = (opcode == `OP_STORE);
assign uload_o = ((opcode == `OP_LOAD) && (func3_o[2] == 1));
assign mem_size_o = (is_load_o || is_store_o)? func3_o[1:0] : `MEM_INV;
assign rd_we_o =(ins_type == `U_TYPE || is_jump_o  || is_load_o || is_op || is_csr_o);

////////////////////////////////////////////////////////////////////////////////
// ENVIRONMENTAL SYSTEM INSTRUCTIONS DECODING (EBREAK / ECALL)
////////////////////////////////////////////////////////////////////////////////
assign isecall = (instr_i == 32'b1110011);
assign isebreak = (instr_i == 32'b00000000000100000000000001110011);
assign system_instr_o = (isecall || isebreak);

////////////////////////////////////////////////////////////////////////////////
// ARITHMETIC LOGIC UNIT (ALU) CONTROL DECODING AND EXPANSIONS
////////////////////////////////////////////////////////////////////////////////
wire is_mul = ((opcode == `OP_OP) && (func7 == 7'b0000001));
reg [4:0] alu_branch_op;
// Combinational evaluation process block mapping the conditional branch operations to the
 corresponding ALU opcode
always @(func3_o) begin
    case (func3_o)
        `F3_BLT:    alu_branch_op <= `ALU_BLT;
        `F3_BLTU:   alu_branch_op <= `ALU_BLTU;
        `F3_BNE:    alu_branch_op <= `ALU_BNE;
        `F3_BGE:    alu_branch_op <= `ALU_BGE;
        `F3_BGEU:   alu_branch_op <= `ALU_BGEU;
        `F3_BEQ:    alu_branch_op <= `ALU_BEQ;
        default:    alu_branch_op <= `ALU_INV;
    endcase
end
wire [4:0] srl_op = (func7[5])? `ALU_SRA : `ALU_SRL;
wire [4:0] addsub_op = (func7 == 7'b0)? `ALU_ADD :
                (opcode == `OP_OPIMM)?  `ALU_ADD :
                                        `ALU_SUB ;

reg [4:0] alu_op_op;
// Combinational evaluation process block routing standard register-register and register-immediate
  operations
always @(func3_o or srl_op or addsub_op) begin
    case (func3_o)
        `F3_ADD:    alu_op_op <= addsub_op;
        `F3_SLT:    alu_op_op <= `ALU_SLT;
        `F3_SLTU:   alu_op_op <= `ALU_SLTU;
        `F3_XOR:    alu_op_op <= `ALU_XOR;
        `F3_OR:     alu_op_op <= `ALU_OR;
        `F3_AND:    alu_op_op <= `ALU_AND;
        `F3_SLL:    alu_op_op <= `ALU_SLL;
        `F3_SRL:    alu_op_op <= srl_op;
    endcase
end

reg [4:0] alu_opmul_op;
// Combinational evaluation process block decoding the RISC-V M-extension hardware multipliers and
```

```
     dividers
    always @(func3_o) begin
        case (func3_o)
            `F3_MUL:    alu_opmul_op <= `ALU_MUL;
            `F3_MULH:   alu_opmul_op <= `ALU_MULH;
            `F3_MULHSU: alu_opmul_op <= `ALU_MULHSU;
            `F3_MULHU:  alu_opmul_op <= `ALU_MULHU;
            `F3_DIV:    alu_opmul_op <= `ALU_DIV;
            `F3_DIVU:   alu_opmul_op <= `ALU_DIVU;
            `F3_REM:    alu_opmul_op <= `ALU_REM;
            `F3_REMU:   alu_opmul_op <= `ALU_REMU;
        endcase
    end

    wire [4:0] alu_opmul_and_op = (is_mul)? alu_opmul_op : alu_op_op;
    // Main execution target routing stage mapping the operational configurations to the master ALU
     output port
    always @(opcode or alu_opmul_and_op or alu_branch_op or alu_op_op) begin
        case (opcode)
            `OP_BRANCH: alu_op_o <= alu_branch_op;
            `OP_OP :    alu_op_o <= alu_opmul_and_op;
            `OP_OPIMM : alu_op_o <= alu_op_op;
            `OP_AUIPC : alu_op_o <= `ALU_ADD;
            `OP_LOAD :  alu_op_o <= `ALU_ADD;
            `OP_STORE : alu_op_o <= `ALU_ADD;
            default:    alu_op_o <= `ALU_INV;
        endcase
    end

    ////////////////////////////////////////////////////////////////////////////////
    // MULTIPLEXER OUTPUT ROUTING FOR IMMEDIATE VECTORS
    ////////////////////////////////////////////////////////////////////////////////
    always @(ins_type or i_immediate or s_immediate or b_immediate or u_immediate or j_immediate )
     begin
        case (ins_type)
            `I_TYPE :   immediate_o <= i_immediate;
            `S_TYPE :   immediate_o <= s_immediate;
            `B_TYPE :   immediate_o <= b_immediate;
            `U_TYPE :   immediate_o <= u_immediate;
            `J_TYPE :   immediate_o <= j_immediate;
            default:    immediate_o <= 32'b0;
        endcase
    end

    ////////////////////////////////////////////////////////////////////////////////
    // ILLEGAL ACCORDANCE AND INTEGRITY CHECKING LOGIC
    ////////////////////////////////////////////////////////////////////////////////
    wire und_ctrl = (ins_type == `UNDEFINED_TYPE);
    wire inv_br = (opcode == `OP_BRANCH) &&
        !((func3_o == `F3_BLT) || (func3_o == `F3_BLTU)|| (func3_o == `F3_BGE) ||
          (func3_o == `F3_BGEU)|| (func3_o == `F3_BNE) || (func3_o == `F3_BEQ));

    wire inv_mem = ((opcode == `OP_STORE) || (opcode == `OP_LOAD)) && (func3_o[1:0] == 2'b11);

    assign illegal_instr_o = (und_ctrl || inv_br || inv_mem);

endmodule
```

cs_registers.v

```
module cs_registers(
    input wire en,              // Global module enable control line
    input wire clk,             // System clock signal
    input wire nrst,            // Asynchronous active-low reset signal
    // CONTROL AND STATUS REGISTER (CSR) INTERFACE PORTS
    input wire is_csr_i,             // Control signal asserted for CSR access instructions
    input wire [2:0] func3_i,        // func3 field extracted from the instruction word
    input wire [31:0] csr_index_i,   // Target CSR memory-mapped address vector (shared with decoder
     immediate)
    input wire [4:0] immediate_i,    // Immediate operand field vector (shared with decoder rs1 field)
    input wire [31:0] rv1_i,         // Data payload read from the Register File source 1 (rs1)
    output reg [31:0] csr_rd_o,      // Output data payload read from the CSR and routed to the
     Register File
    // ASYNCHRONOUS INTERRUPT INTERFACE PORTS
    input wire external_interrupt_i, // Asynchronous machine external interrupt request line
    // SYNCHRONOUS EXCEPTION INTERFACE PORTS
    input wire illegal_instr_i,      // Exception flag asserted upon an illegal opcode condition from
     the Decoder stage
    input wire system_instr_i,       // Control signal asserted for environmental system instructions (
     ECALL/EBREAK)
    input wire instr_misaligned_i,   // Exception flag asserted for instruction address misaligned
     conditions (PC[1:0] != 2'b00)
```

```
    input wire misaligned_load_i,    // Exception flag asserted for data address misaligned conditions
     during a load operation
    input wire misaligned_store_i,   // Exception flag asserted for data address misaligned conditions
     during a store operation
    // TRAP AND HANDLER CONTROL PORTS
    input wire [31:0] pc_i,              // Current Program Counter (PC) to be saved into mepc upon
     trap assertion
    input wire is_mret_i,                // Control signal asserted for MRET (Machine Trap Return)
     instructions
    output wire enter_trap_o,            // Trap assertion flag indicating the core should vector to
     the trap handler
    output wire exit_trap_o,             // Trap return flag indicating execution context restoration (
     mret detection)
    output wire [31:0] return_address_o, // Context restoration target address bus routed from mepc
    output wire [31:0] trap_address_o    // Trap vector base address bus routed from mtvec
    );

    ////////////////////////////////////////////////////////////////////////////////
    // Control and Status Registers Architectural Initialization Defaults
    ////////////////////////////////////////////////////////////////////////////////
    localparam DEF_MIE_MEIE = 1'b1;
    // Trap Vector Base Address and Mode Configuration Constraints
    localparam MTVEC_BASE = 30'h1c;     // Base vector address for exception handlers
    localparam MTVEC_MODE = 2'b0;       // Vectoring Mode selection (00: Direct, 01: Vectored, 10/11:
     Reserved)

    ////////////////////////////////////////////////////////////////////////////////
    // RISC-V Architectural Cause Encoding Definitions (mcause)
    ////////////////////////////////////////////////////////////////////////////////
    // Asynchronous Trap Causes (Interrupts => mcause[31] = 1)
    localparam MACHINE_SOFTWARE_INTERRUPT = 3,
               MACHINE_TIMER_INTERRUPT    = 7,
               MACHINE_EXTERNAL_INTERRUPT = 11,
    // Synchronous Trap Causes (Exceptions => mcause[31] = 0)
               INSTRUCTION_ADDRESS_MISALIGNED = 0,
               ILLEGAL_INSTRUCTION            = 2,
               EBREAK                         = 3,
               LOAD_ADDRESS_MISALIGNED        = 4,
               STORE_ADDRESS_MISALIGNED       = 6,
               ECALL                          = 11;

    ////////////////////////////////////////////////////////////////////////////////
    // Architectural Control and Status Register File Declarations
    ////////////////////////////////////////////////////////////////////////////////
    reg [31:0] mepc;                // Machine Exception Program Counter: stores the trap return
     address
    reg [31:0] mtvec;               // Machine Trap Vector base address and operating mode register
    reg mcause_int;                 // Intercepted trap classification flag (1: Asynchronous Interrupt,
      0: Synchronous Exception)
    reg [3:0] mcause_code;          // Encoded trap identifier tracking specific interrupt or exception
      causes
    reg mstatus_mie;                // Machine Interrupt Enable status bit (global control line)
    reg mstatus_mpie;               // Machine Previous Interrupt Enable status bit (latches MIE state
     upon trap)
    reg [1:0] mstatus_mpp;          // Machine Previous Privilege mode (hardwired to 2'b11 Machine-mode
      in this architecture)
    reg mie_meie;                   // Machine External Interrupt Enable control bit

    // Synchronous pulse edge detection logic for asynchronous external interrupt filtering
    reg external_interrupt_prev;
    always @(posedge clk, negedge nrst) begin
        if (!nrst)
            external_interrupt_prev <= 1'b0;
        else
            external_interrupt_prev <= external_interrupt_i;
    end

    // Machine Interrupt Pending (mip) field tracking for external line state transitions
    wire mip_meie = !external_interrupt_prev & external_interrupt_i;
    // Continuous assignment for architectural context status routing
    assign return_address_o = mepc;
    assign trap_address_o = mtvec;

    // Interrupt evaluation gating logic evaluating qualified pending interrupts
    wire external_int_pending = ((mip_meie) && (mie_meie) && (nrst));
    wire is_interrupt = external_int_pending;
    wire is_exception = (instr_misaligned_i || illegal_instr_i || system_instr_i ||
                         misaligned_load_i     || misaligned_store_i);
    // Operational trap evaluation gating outputs
    assign enter_trap_o = is_interrupt || is_exception;
    assign exit_trap_o = is_mret_i;

    // Hardware structural concatenation layouts for structural CSR read access mapping
    wire [31:0] mstatus_reg = {19'b0, mstatus_mpp, 3'b0, mstatus_mpie, 3'b0, mstatus_mie, 3'b0};
    wire [31:0] mie_reg     = {20'b0, mie_meie, 11'b0};
    wire [31:0] mip_reg     = {20'b0, mip_meie, 11'b0};
    wire [31:0] mcause_reg  = {mcause_int, 27'b0 , mcause_code};
```

```
// Combinational multiplexing evaluation process block for target CSR read bus operations
always @(csr_index_i or mstatus_reg or mie_reg or mtvec or mepc or mcause_reg or mip_reg ) begin
    case (csr_index_i)
        `MSTATUS :  csr_rd_o <= mstatus_reg;
        `MIE :      csr_rd_o <= mie_reg;
        `MTVEC :    csr_rd_o <= mtvec;
        `MEPC :     csr_rd_o <= mepc;
        `MCAUSE :   csr_rd_o <= mcause_reg;
        `MIP :      csr_rd_o <= mip_reg;
        default :   csr_rd_o <= 32'b0;
    endcase
end

// Combinational evaluation process block executing atomic operations on targeted CSR payloads
reg [31:0] csr_in;
always @(func3_i or rv1_i or csr_rd_o or immediate_i ) begin
    case (func3_i)
        `F3_RW :    csr_in <= rv1_i;
        `F3_RS :    csr_in <= rv1_i | csr_rd_o;
        `F3_RC :    csr_in <= (~rv1_i) & csr_rd_o;
        `F3_RWI :   csr_in <= immediate_i;
        `F3_RSI :   csr_in <= immediate_i | csr_rd_o;
        `F3_RCI :   csr_in <= immediate_i & (~csr_rd_o);
        default :   csr_in <= 32'b0;
    endcase
end

// Synchronous process block handling atomic CSR updates, trap vectoring sequences, and context
 restoration
always @(posedge clk, negedge nrst) begin
    if (!nrst) begin
        mepc        <= 32'b0;
        mtvec       <= {MTVEC_BASE, MTVEC_MODE};
        mcause_int  <= 1'b0;
        mcause_code <= 4'b0;
        mstatus_mpp  <= 2'b11;
        mstatus_mpie <= 1'b0;
        mstatus_mie  <= 1'b0;
        mie_meie    <= DEF_MIE_MEIE;
    end
    else if (en) begin
        // ENTER TRAP HANDLING SEQUENCE
        if (enter_trap_o) begin
            mstatus_mpp  <= 2'b11;
            mstatus_mpie <= mstatus_mie;
            mstatus_mie  <= 1'b0;
            mepc         <= pc_i;
            if(is_interrupt) begin
                mcause_int <= 1'b1;
                if (external_int_pending) begin
                    mcause_code <= MACHINE_EXTERNAL_INTERRUPT;
                end
            end
            else begin
                mcause_int <= 1'b0;
                if (illegal_instr_i)
                    mcause_code <= ILLEGAL_INSTRUCTION;
                else if (instr_misaligned_i)
                    mcause_code <= INSTRUCTION_ADDRESS_MISALIGNED;
                else if (system_instr_i)
                    mcause_code <= EBREAK;
                else if (misaligned_load_i)
                    mcause_code <= LOAD_ADDRESS_MISALIGNED;
                else if (misaligned_store_i)
                    mcause_code <= STORE_ADDRESS_MISALIGNED;
            end
        end
        // EXIT TRAP HANDLING SEQUENCE (MRET EXECUTION CONTEXT RESTORATION)
        else if (exit_trap_o) begin
            mstatus_mpp  <= 2'b11;
            mstatus_mie  <= mstatus_mpie;
            mstatus_mpie <= 1'b1;
        end
        // SYNCHRONOUS ATOMIC WRITE/MODIFY OPERATION ROUTING
        else if (is_csr_i) begin
            case (csr_index_i)
                `MSTATUS :  begin
                                mstatus_mie  <= csr_in[3];
                                mstatus_mpie <= csr_in[7];
                            end
                `MIE :      begin
                                mie_meie <= csr_in[11];
                            end
                `MTVEC :    begin
                                mtvec <= csr_in;
                            end
```

```
                    `MEPC :     begin
                                    mepc <= {csr_in[31:2],2'b0};
                                end
                    `MCAUSE :   begin
                                    mcause_int  <= csr_in[31];
                                    mcause_code <= csr_in[3:0];
                                end
                endcase
            end
        end
    end
endmodule
```

alu.v

```
module alu(
    input wire [31:0] value1_i,         // Primary execution operand input 1 for the ALU subsystems
    input wire [31:0] value2_i,         // Secondary execution operand input 2 for the ALU subsystems
    input wire [31:0] immediate_i,      // Extended immediate value vector formatted by the Instruction
      Decoder stage
    input wire [31:0] pc_i,             // Current Program Counter (PC) register value
    input wire [4:0] alu_op_i,          // 5-bit control opcode defining the master ALU operation
    input wire is_jalr_i,               // Control signal asserted when the execution instruction is a
     JALR
    output wire [31:0] adder_result_o,  // Unformatted output bus from the primary arithmetic adder
    output wire branch_result_o,        // Evaluation result flag generated by the conditional branch
     comparison logic
    output reg [31:0] result_o,         // Multiplexed master data output bus of the ALU execution
     stage
    output wire [31:0] next_pc_o        // Calculated target Program Counter (PC) address bus
    );

    ////////////////////////////////////////////////////////////////////////////////
    // PRIMARY ARITHMETIC ADDER SYSTEM
    ////////////////////////////////////////////////////////////////////////////////
    wire [32:0] adder_in1 = {value1_i,1'b1};
    wire op2_neg = ((alu_op_i == `ALU_SUB)  || (alu_op_i == `ALU_BEQ)  || (alu_op_i == `ALU_BNE)
                 || (alu_op_i == `ALU_BGE)  || (alu_op_i == `ALU_BGEU) || (alu_op_i == `ALU_BLT)
                 || (alu_op_i == `ALU_BLTU) || (alu_op_i == `ALU_SLT)  || (alu_op_i == `ALU_SLTU))? 1'
     b1 : 1'b0;

    wire [32:0] value2_neg = {value2_i,1'b0} ^ {33{1'b1}};
    wire [32:0] adder_in2 = op2_neg? value2_neg : {value2_i, 1'b0};
    wire [32:0] adder_result = $unsigned(adder_in1) + $unsigned(adder_in2);
        assign adder_result_o = adder_result[32:1];

    ////////////////////////////////////////////////////////////////////////////////
    // CONDITIONAL BRANCH EVALUATION AND COMPARISON LOGIC
    ////////////////////////////////////////////////////////////////////////////////
    wire signed_op = ((alu_op_i == `ALU_BGE) || (alu_op_i == `ALU_BLT) || (alu_op_i == `ALU_SLT) )? 1'
     b1 : 1'b0;
    wire is_equal = (adder_result == 32'b0)? 1'b1 : 1'b0;
    wire is_greater_equal = ((value1_i[31]^value2_i[31]) == 1'b0)? (adder_result[32] == 1'b0) : (
     value1_i[31] ^ (signed_op));
    reg branch_result;

    // Combinational evaluation process block resolving conditional branches and set-less-than
     comparisons
    always @(is_equal or is_greater_equal or alu_op_i) begin
        case (alu_op_i)
            `ALU_BEQ :  branch_result <= is_equal;
            `ALU_BNE :  branch_result <= ~is_equal;
            `ALU_BGE :  branch_result <= is_greater_equal;
            `ALU_BGEU : branch_result <= is_greater_equal;
            `ALU_BLT :  branch_result <= ~is_greater_equal;
            `ALU_BLTU : branch_result <= ~is_greater_equal;
            `ALU_SLT :  branch_result <= ~is_greater_equal;
            `ALU_SLTU : branch_result <= ~is_greater_equal;
            default :   branch_result <= 1'b0;
        endcase
    end

    // Gating logic to isolate branch outputs from Set Less Than (SLT/SLTU) tracking flags
    assign branch_result_o = ((alu_op_i == `ALU_SLT) || (alu_op_i == `ALU_SLTU))?   1'b0 :
     branch_result;

    ////////////////////////////////////////////////////////////////////////////////
    // BITWISE LOGICAL OPERATIONS
    ////////////////////////////////////////////////////////////////////////////////
    wire [31:0] or_result  = value1_i | value2_i;
    wire [31:0] and_result = value1_i & value2_i;
```

```
    wire [31:0] xor_result = value1_i ^ value2_i;
    reg [31:0] bwlogic_result;
    // Combinational evaluation process block routing bitwise operations based on opcode selection
    always @(or_result or and_result or xor_result or alu_op_i) begin
        case (alu_op_i)
            `ALU_OR :   bwlogic_result <= or_result;
            `ALU_AND :  bwlogic_result <= and_result;
            `ALU_XOR :  bwlogic_result <= xor_result;
            default :   bwlogic_result <= 32'b0;
        endcase
    end

    ////////////////////////////////////////////////////////////////////////////////
    // BARREL SHIFTER HARDWARE DATAPATHS
    ////////////////////////////////////////////////////////////////////////////////
    reg [31:0] shift_result;
    // Combinational evaluation process block executing logical/arithmetic shift steering operations
    always @(alu_op_i or value1_i or value2_i) begin
        case (alu_op_i)
            `ALU_SLL :  shift_result <= value1_i << value2_i[4:0];
            `ALU_SRL :  shift_result <= value1_i >> value2_i[4:0];
            `ALU_SRA :  shift_result <= $signed(value1_i) >>> value2_i[4:0];
            default :   shift_result <= 32'b0;
        endcase
    end

    ////////////////////////////////////////////////////////////////////////////////
    // ALU MASTER OUTPUT BUS MULTIPLEXINGSTAGE
    ////////////////////////////////////////////////////////////////////////////////
    always @(alu_op_i or bwlogic_result or adder_result_o or shift_result or branch_result ) begin
        case (alu_op_i)
            `ALU_XOR :  result_o <= bwlogic_result;
            `ALU_OR :   result_o <= bwlogic_result;
            `ALU_AND :  result_o <= bwlogic_result;
            `ALU_ADD :  result_o <= adder_result_o;
            `ALU_SUB :  result_o <= adder_result_o;
            `ALU_SLL :  result_o <= shift_result;
            `ALU_SRL :  result_o <= shift_result;
            `ALU_SRA :  result_o <= shift_result;
            default :   result_o <=  {{31{1'b0}},branch_result};
        endcase
    end

    ////////////////////////////////////////////////////////////////////////////////
    // NEXT PROGRAM COUNTER (PC) TARGET RESOLUTION LOGIC
    ////////////////////////////////////////////////////////////////////////////////
    assign next_pc_o = is_jalr_i? value1_i + immediate_i : pc_i + immediate_i;

endmodule
```

Appendix B

Firmware for Microprocessor Characterization

Table B.1: Firmware characterization loop for the *ADD* (*R-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
37B08093	`addi x1, x1, 891`	Add Imm 891 (0x37B) to register x1.
7FF10113	`addi x2, x2, 2047`	Add Imm 2047 (0x7FF) to register x2.
70F10113	`addi x2, x2, 1807`	Add Imm 1807 (0x70F) to register x2.
002080B3	`add x1, x1, x2`	Add register x2 to x1; store result in register x1.
01F080B3	`add x1, x1, x31`	Add register x31 to x1; store result in register x1.
001080B3	`add x1, x1, x1`	Add register x1 to x1; store result in register x1.
01F08FB3	`add x31, x1, x31`	Add register x31 to x1; store result in register x31.
00208FB3	`add x31, x1, x2`	Add register x2 to x1; store result in register x31.
01F18FB3	`add x31, x3, x31`	Add register x31 to x3; store result in register x31.
00110133	`add x4, x18, x1`	Add register x1 to x18; store result in register x4.
01F10133	`add x4, x18, x31`	Add register x31 to x18; store result in register x4.
01F080B3	`add x1, x1, x31`	Add register x31 to x1; store result in register x1.
002080B3	`add x1, x1, x2`	Add register x2 to x1; store result in register x1.
00110133	`add x4, x18, x1`	Add register x1 to x18; store result in register x4.
01110133	`add x4, x18, x17`	Add register x17 to x18; store result in register x4.
01F08FB3	`add x31, x1, x31`	Add register x31 to x1; store result in register x31.
00208FB3	`add x31, x1, x2`	Add register x2 to x1; store result in register x31.
01F18FB3	`add x31, x3, x31`	Add register x31 to x3; store result in register x31.
00110133	`add x4, x18, x1`	Add register x1 to x18; store result in register x4.
01F10133	`add x4, x18, x31`	Add register x31 to x18; store result in register x4.
01F080B3	`add x1, x1, x31`	Add register x31 to x1; store result in register x1.
002080B3	`add x1, x1, x2`	Add register x2 to x1; store result in register x1.
01F08FB3	`add x31, x1, x31`	Add register x31 to x1; store result in register x31.
00208FB3	`add x31, x1, x2`	Add register x2 to x1; store result in register x31.
01F18FB3	`add x31, x3, x31`	Add register x31 to x3; store result in register x31.
00110133	`add x4, x18, x1`	Add register x1 to x18; store result in register x4.
01F10133	`add x4, x18, x31`	Add register x31 to x18; store result in register x4.
01F080B3	`add x1, x1, x31`	Add register x31 to x1; store result in register x1.
002080B3	`add x1, x1, x2`	Add register x2 to x1; store result in register x1.
01F08FB3	`add x31, x1, x31`	Add register x31 to x1; store result in register x31.
00208FB3	`add x31, x1, x2`	Add register x2 to x1; store result in register x31.
F91FF06F	`jal x0, -110`	Unconditional relative jump to PC - 110.

Table B.2: Firmware characterization loop for the $ADDI$ (I-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
03908093	addi x1, x1, 57	Add Imm 57 (0x39) to register x1.
00210113	addi x2, x2, 2	Add Imm value 2 to register x2.
00318193	addi x3, x3, 3	Add Imm value 3 to register x3.
0B220213	addi x4, x4, 178	Add Imm value 178 to register x4.
00528293	addi x5, x5, 5	Add Imm value 5 to register x5.
00630313	addi x6, x6, 6	Add Imm value 6 to register x6.
26E18193	addi x3, x3, 622	Add Imm value 622 to register x3.
00840413	addi x8, x8, 8	Add Imm value 8 to register x8.
00948493	addi x9, x9, 9	Add Imm value 9 to register x9.
00A50513	addi x10, x10, 10	Add Imm value 10 to register x10.
00B58593	addi x11, x11, 11	Add Imm value 11 to register x11.
51A28293	addi x5, x5, 1306	Add Imm value 1306 to register x5.
00D68693	addi x13, x13, 13	Add Imm value 13 to register x13.
00E70713	addi x14, x14, 14	Add Imm value 14 to register x14.
51A28293	addi x5, x5, 1306	Add Imm value 1306 to register x5.
01080813	addi x16, x16, 16	Add Imm value 16 to register x16.
73868693	addi x13, x13, 1848	Add Imm value 1848 to register x13.
01290913	addi x17, x17, 18	Add Imm value 18 to register x17.
13298993	addi x9, x19, 306	Add Imm value 306 to x19; store result in x9.
1B4A0A13	addi x20, x20, 436	Add Imm value 436 to register x20.
2AE28293	addi x5, x30, 686	Add Imm value 686 to x30; store result in x5.
016B0B13	addi x22, x22, 22	Add Imm value 22 to register x22.
1B7B8B13	addi x23, x23, 439	Add Imm value 439 to register x23.
018C0C13	addi x24, x24, 24	Add Imm value 24 to register x24.
019C8C13	addi x25, x25, 25	Add Imm value 25 to register x25.
01AD0D13	addi x26, x26, 26	Add Imm value 26 to register x26.
01BD8D13	addi x27, x27, 27	Add Imm value 27 to register x27.
2AE28293	addi x5, x30, 686	Add Imm value 686 to x30; store result in x5.
1BD68693	addi x13, x13, 445	Add Imm value 445 to register x13.
1BE70713	addi x14, x14, 446	Add Imm value 446 to register x14.
01F58593	addi x11, x11, 31	Add Imm value 31 to register x11.
F85FF06F	jal x0, -120	Unconditional relative jump to PC - 120.

Table B.3: Firmware characterization loop for the AND (R-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
37B08093	addi x1, x1, 891	Add Imm 891 (0x37B) to register x1.
7FF10113	addi x2, x2, 2047	Add Imm 2047 (0x7FF) to register x2.
70F18193	addi x3, x3, 1807	Add Imm 1807 (0x70F) to register x3.
00C1F1B3	and x3, x3, x12	AND between x3 and x12; store in register x3.
00D27233	and x4, x4, x13	AND between x4 and x13; store in register x4.
00E2F2B3	and x5, x5, x14	AND between x5 and x14; store in register x5.
00F37333	and x6, x6, x15	AND between x6 and x15; store in register x6.
0011F3B3	and x7, x3, x1	AND between x3 and x1; store in register x7.
00227433	and x8, x4, x2	AND between x4 and x2; store in register x8.
0032F4B3	and x9, x5, x3	AND between x5 and x3; store in register x9.
00437533	and x10, x6, x4	AND between x6 and x4; store in register x10.
0051F5B3	and x11, x3, x5	AND between x3 and x5; store in register x11.
00627633	and x12, x4, x6	AND between x4 and x6; store in register x12.
0072F6B3	and x13, x5, x7	AND between x5 and x7; store in register x13.
00837733	and x14, x6, x8	AND between x6 and x8; store in register x14.
0091F7B3	and x15, x3, x9	AND between x3 and x9; store in register x15.
00A27833	and x16, x4, x10	AND between x4 and x10; store in register x16.
00B2F8B3	and x17, x5, x11	AND between x5 and x11; store in register x17.
00C37933	and x18, x6, x12	AND between x6 and x12; store in register x18.
00D1F9B3	and x19, x3, x13	AND between x3 and x13; store in register x19.
00E27A33	and x20, x4, x14	AND between x4 and x14; store in register x20.
00F2FAB3	and x21, x5, x15	AND between x5 and x15; store in register x21.
00137B33	and x22, x6, x1	AND between x6 and x1; store in register x22.
0021FB33	and x23, x3, x2	AND between x3 and x2; store in register x23.
00327C33	and x24, x4, x3	AND between x4 and x3; store in register x24.
0042FCB3	and x25, x5, x4	AND between x5 and x4; store in register x25.
00537D33	and x26, x6, x5	AND between x6 and x5; store in register x26.
0061FDB3	and x27, x3, x6	AND between x3 and x6; store in register x27.
00727E33	and x28, x4, x7	AND between x4 and x7; store in register x28.
0082FEB3	and x29, x5, x8	AND between x5 and x8; store in register x29.
0011F133	and x2, x3, x1	AND between x3 and x1; store in register x2.
F91FF06F	jal x0, -110	Unconditional relative jump to PC - 110.

Table B.4: Firmware characterization loop for the $ANDI$ (*I-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
F9CF7113	andi x2, x31, -100	AND between x31 and Imm -100; store in x2.
37B0F093	andi x1, x1, 891	AND between x1 and Imm 891; store in x1.
F9CF7113	andi x2, x31, -100	AND between x31 and Imm -100; store in x2.
7FF17113	andi x2, x2, 2047	AND between x2 and Imm 2047; store in x2.
F9CF7113	andi x2, x31, -100	AND between x31 and Imm -100; store in x2.
1FF1F093	andi x1, x3, 511	AND between x3 and Imm 511; store in x1.
F9CF7113	andi x2, x31, -100	AND between x31 and Imm -100; store in x2.
07B0F093	andi x1, x1, 123	AND between x1 and Imm 123; store in x1.
F9CF7113	andi x2, x31, -100	AND between x31 and Imm -100; store in x2.
0AA17113	andi x2, x2, 170	AND between x2 and Imm 170; store in x2.
2480FF13	andi x30, x1, 584	AND between x1 and Imm 584; store in x30.
F9CF7113	andi x2, x31, -100	AND between x31 and Imm -100; store in x2.
3841FF13	andi x30, x2, 900	AND between x2 and Imm 900; store in x30.
190F7093	andi x9, x30, 400	AND between x30 and Imm 400; store in x9.
E0017F13	andi x30, x2, -512	AND between x2 and Imm -512; store in x30.
00AF7193	andi x3, x30, 10	AND between x30 and Imm 10; store in x3.
7D00FF13	andi x30, x1, 2000	AND between x1 and Imm 2000; store in x30.
00CF7113	andi x2, x30, 12	AND between x30 and Imm 12; store in x2.
3091FF13	andi x30, x2, 777	AND between x2 and Imm 777; store in x30.
800F7093	andi x9, x30, -2048	AND between x30 and Imm -2048; store in x9.
06F17F13	andi x30, x2, 111	AND between x2 and Imm 111; store in x30.
378F7193	andi x3, x30, 888	AND between x30 and Imm 888; store in x3.
02C0FF13	andi x30, x1, 44	AND between x1 and Imm 44; store in x30.
3E7F7113	andi x2, x30, 999	AND between x30 and Imm 999; store in x2.
0011FF13	andi x30, x2, 1	AND between x2 and Imm 1; store in x30.
3FFF7093	andi x9, x30, 1023	AND between x30 and Imm 1023; store in x9.
00F17F13	andi x30, x2, 15	AND between x2 and Imm 15; store in x30.
037F7193	andi x3, x30, 55	AND between x30 and Imm 55; store in x3.
0000FF13	andi x30, x1, 0	AND between x1 and Imm 0; store in x30.
FFEF7113	andi x2, x30, -17	AND between x30 and Imm -17; store in x2.
4D21FF13	andi x30, x2, 1234	AND between x2 and Imm 1234; store in x30.
F85FF06F	jal x0, -124	Unconditional relative jump backward to PC - 124.

Table B.5: Firmware characterization loop for the *AUIPC* (*U-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00FFF617	auipc x12, 0x00FFF	Add 0x00FFF000 to PC; store address in x12.
33330C17	auipc x24, 0x33330	Add 0x33330000 to PC; store address in x24.
3FFFF917	auipc x18, 0x3FFFF	Add 0x3FFFF000 to PC; store address in x18.
33330C17	auipc x24, 0x33330	Add 0x33330000 to PC; store address in x24.
00003117	auipc x2, 0x00003	Add 0x00003000 to PC; store address in x2.
00007197	auipc x3, 0x00007	Add 0x00007000 to PC; store address in x3.
0000F217	auipc x4, 0x0000F	Add 0x0000F000 to PC; store address in x4.
0001F297	auipc x5, 0x0001F	Add 0x0001F000 to PC; store address in x5.
0003F317	auipc x6, 0x0003F	Add 0x0003F000 to PC; store address in x6.
0FF00E17	auipc x28, 0x0FF00	Add 0x0FF00000 to PC; store address in x28.
000FF417	auipc x28, 0x000FF	Add 0x000FF000 to PC; store address in x28.
001FF497	auipc x29, 0x001FF	Add 0x001FF000 to PC; store address in x29.
33330C17	auipc x24, 0x33330	Add 0x33330000 to PC; store address in x24.
3FFFF917	auipc x18, 0x3FFFF	Add 0x3FFFF000 to PC; store address in x18.
00FFF617	auipc x12, 0x00FFF	Add 0x00FFF000 to PC; store address in x12.
01FFF697	auipc x13, 0x01FFF	Add 0x01FFF000 to PC; store address in x13.
03FFF717	auipc x14, 0x03FFF	Add 0x03FFF000 to PC; store address in x14.
FF000C97	auipc x25, -4096	Add 0xFF000000 to PC; store address in x25.
0FF00E17	auipc x28, 0x0FF00	Add 0x0FF00000 to PC; store address in x28.
1FFFF897	auipc x9, 0x1FFFF	Add 0x1FFFF000 to PC; store address in x9.
3FFFF917	auipc x18, 0x3FFFF	Add 0x3FFFF000 to PC; store address in x18.
7FFFFA17	auipc x20, 0x7FFFF	Add 0x7FFFF000 to PC; store address in x20.
FFFFFB17	auipc x22, -1	Add 0xFFFFF000 to PC; store address in x22.
007FF597	auipc x11, 0x007FF	Add 0x007FF000 to PC; store address in x11.
0FF00E17	auipc x28, 0x0FF00	Add 0x0FF00000 to PC; store address in x28.
01FFF697	auipc x13, 0x01FFF	Add 0x01FFF000 to PC; store address in x13.
33330C17	auipc x24, 0x33330	Add 0x33330000 to PC; store address in x24.
FF000C97	auipc x25, -4096	Add 0xFF000000 to PC; store address in x25.
00FF0D17	auipc x10, 0x00FF0	Add 0x00FF0000 to PC; store address in x10.
F00F0D97	auipc x11, -65296	Add 0xF00F0000 to PC; store address in x11.
0FF00E17	auipc x28, 0x0FF00	Add 0x0FF00000 to PC; store address in x28.
F85FF06F	jal x0, -124	Unconditional relative jump backward to PC - 124.

Table B.6: Firmware characterization loop for the *BEQ* (*B-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00100093	`addi x1, x0, 1`	Add Imm 1 to x0; store result in register x1.
00000113	`addi x2, x0, 0`	Load Imm value 0 into register x2 (li pseudo-op).
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00108463	`beq x1, x1, 8`	Branch to PC + 8 if x1 equals x1.
00000000	`nop`	No operation.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
00208463	`beq x1, x2, 8`	Branch to PC + 8 if register x1 equals register x2.
F81086E3	`beq x1, x1, -116`	Branch back to PC - 116 if register x1 equals x1.

Table B.7: Firmware characterization loop for the *BGE* (*B-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00100093	`addi x1, x0, 1`	Add *Imm* 1 to x0; store result in register x1.
00000113	`li x2, 0`	Load *Imm* value 0 into register x2.
0010d463	`bge x1, x1, 8`	Branch to PC + 8 if x1 is greater than or equal to x1.
00000013	`nop`	No operation (implemented as addi x0, x0, 0).
0010d463	`bge x1, x1, 8`	Branch to PC + 8 if x1 is greater than or equal to x1.
00000000	`nop`	No operation.
0010d463	`bge x1, x1, 8`	Branch to PC + 8 if x1 is greater than or equal to x1.
00000000	`nop`	No operation.
0010d463	`bge x1, x1, 8`	Branch to PC + 8 if x1 is greater than or equal to x1.
00000000	`nop`	No operation.
0010d463	`bge x1, x1, 8`	Branch to PC + 8 if x1 is greater than or equal to x1.
00000000	`nop`	No operation.
0010d463	`bge x1, x1, 8`	Branch to PC + 8 if x1 is greater than or equal to x1.
00000000	`nop`	No operation.
0010d463	`bge x1, x1, 8`	Branch to PC + 8 if x1 is greater than or equal to x1.
00000000	`nop`	No operation.
0010d463	`bge x1, x1, 8`	Branch to PC + 8 if x1 is greater than or equal to x1.
00000000	`nop`	No operation.
0010d463	`bge x1, x1, 8`	Branch to PC + 8 if x1 is greater than or equal to x1.
00000000	`nop`	No operation.
00115463	`bge x2, x1, 8`	Branch to PC + 8 if x2 is greater than or equal to x1.
00115463	`bge x2, x1, 8`	Branch to PC + 8 if x2 is greater than or equal to x1.
00115463	`bge x2, x1, 8`	Branch to PC + 8 if x2 is greater than or equal to x1.
00115463	`bge x2, x1, 8`	Branch to PC + 8 if x2 is greater than or equal to x1.
00115463	`bge x2, x1, 8`	Branch to PC + 8 if x2 is greater than or equal to x1.
00115463	`bge x2, x1, 8`	Branch to PC + 8 if x2 is greater than or equal to x1.
00115463	`bge x2, x1, 8`	Branch to PC + 8 if x2 is greater than or equal to x1.
00115463	`bge x2, x1, 8`	Branch to PC + 8 if x2 is greater than or equal to x1.
00115463	`bge x2, x1, 8`	Branch to PC + 8 if x2 is greater than or equal to x1.
00115463	`bge x2, x1, 8`	Branch to PC + 8 if x2 is greater than or equal to x1.
00000000	`nop`	No operation.
f810d6e3	`bge x1, x1, -116`	Branch to PC - 116 if x1 is greater than or equal to x1.

Table B.8: Firmware characterization loop for the *BGEU* (*B-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00100093	`addi x1, x0, 1`	Add Imm 1 to x0; store result in register x1.
00000113	`li x2, 0`	Load Imm value 0 into register x2.
0010F463	`bgeu x1, x1, 8`	Branch to PC + 8 if x1 ¿= x1 (unsigned comparison).
00000000	`nop`	No operation.
0010F463	`bgeu x1, x1, 8`	Branch to PC + 8 if x1 ¿= x1 (unsigned comparison).
00000000	`nop`	No operation.
0010F463	`bgeu x1, x1, 8`	Branch to PC + 8 if x1 ¿= x1 (unsigned comparison).
00000000	`nop`	No operation.
0010F463	`bgeu x1, x1, 8`	Branch to PC + 8 if x1 ¿= x1 (unsigned comparison).
00000000	`nop`	No operation.
0010F463	`bgeu x1, x1, 8`	Branch to PC + 8 if x1 ¿= x1 (unsigned comparison).
00000000	`nop`	No operation.
0010F463	`bgeu x1, x1, 8`	Branch to PC + 8 if x1 ¿= x1 (unsigned comparison).
00000000	`nop`	No operation.
0010F463	`bgeu x1, x1, 8`	Branch to PC + 8 if x1 ¿= x1 (unsigned comparison).
00000000	`nop`	No operation.
0010F463	`bgeu x1, x1, 8`	Branch to PC + 8 if x1 ¿= x1 (unsigned comparison).
00000000	`nop`	No operation.
0010F463	`bgeu x1, x1, 8`	Branch to PC + 8 if x1 ¿= x1 (unsigned comparison).
00000000	`nop`	No operation.
0020F463	`bgeu x1, x2, 8`	Branch to PC + 8 if x1 ¿= x2 (unsigned comparison).
0020F463	`bgeu x1, x2, 8`	Branch to PC + 8 if x1 ¿= x2 (unsigned comparison).
0020F463	`bgeu x1, x2, 8`	Branch to PC + 8 if x1 ¿= x2 (unsigned comparison).
0020F463	`bgeu x1, x2, 8`	Branch to PC + 8 if x1 ¿= x2 (unsigned comparison).
0020F463	`bgeu x1, x2, 8`	Branch to PC + 8 if x1 ¿= x2 (unsigned comparison).
0020F463	`bgeu x1, x2, 8`	Branch to PC + 8 if x1 ¿= x2 (unsigned comparison).
0020F463	`bgeu x1, x2, 8`	Branch to PC + 8 if x1 ¿= x2 (unsigned comparison).
0020F463	`bgeu x1, x2, 8`	Branch to PC + 8 if x1 ¿= x2 (unsigned comparison).
0020F463	`bgeu x1, x2, 8`	Branch to PC + 8 if x1 ¿= x2 (unsigned comparison).
0020F463	`bgeu x1, x2, 8`	Branch to PC + 8 if x1 ¿= x2 (unsigned comparison).
00000000	`nop`	No operation.
F810F6E3	`bgeu x1, x1, -116`	Branch to PC - 116 if x1 ¿= x1 (unsigned loop).

Table B.9: Firmware characterization loop for the BLT (B-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00100093	`addi x1, x0, 1`	Add Imm 1 to x0; store result in register x1.
00000113	`li x2, 0`	Load Imm value 0 into register x2.
00114463	`blt x2, x1, 8`	Branch to PC + 8 if x2 is less than x1.
00000000	`nop`	No operation.
00114463	`blt x2, x1, 8`	Branch to PC + 8 if x2 is less than x1.
00000000	`nop`	No operation.
00114463	`blt x2, x1, 8`	Branch to PC + 8 if x2 is less than x1.
00000000	`nop`	No operation.
00114463	`blt x2, x1, 8`	Branch to PC + 8 if x2 is less than x1.
00000000	`nop`	No operation.
00114463	`blt x2, x1, 8`	Branch to PC + 8 if x2 is less than x1.
00000000	`nop`	No operation.
00114463	`blt x2, x1, 8`	Branch to PC + 8 if x2 is less than x1.
00000000	`nop`	No operation.
00114463	`blt x2, x1, 8`	Branch to PC + 8 if x2 is less than x1.
00000000	`nop`	No operation.
00114463	`blt x2, x1, 8`	Branch to PC + 8 if x2 is less than x1.
00000000	`nop`	No operation.
00114463	`blt x2, x1, 8`	Branch to PC + 8 if x2 is less than x1.
00000000	`nop`	No operation.
0020C463	`blt x1, x2, 8`	Branch to PC + 8 if x1 is less than x2.
0020C463	`blt x1, x2, 8`	Branch to PC + 8 if x1 is less than x2.
0020C463	`blt x1, x2, 8`	Branch to PC + 8 if x1 is less than x2.
0020C463	`blt x1, x2, 8`	Branch to PC + 8 if x1 is less than x2.
0020C463	`blt x1, x2, 8`	Branch to PC + 8 if x1 is less than x2.
0020C463	`blt x1, x2, 8`	Branch to PC + 8 if x1 is less than x2.
0020C463	`blt x1, x2, 8`	Branch to PC + 8 if x1 is less than x2.
0020C463	`blt x1, x2, 8`	Branch to PC + 8 if x1 is less than x2.
0020C463	`blt x1, x2, 8`	Branch to PC + 8 if x1 is less than x2.
0020C463	`blt x1, x2, 8`	Branch to PC + 8 if x1 is less than x2.
00000000	`nop`	No operation.
F81146E3	`blt x2, x1, -116`	Branch to PC - 116 if x2 is less than x1.

Table B.10: Firmware characterization loop for the *BLTU* (*B-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00100093	addi x1, x0, 1	Add Imm 1 to x0; store result in register x1.
00000113	li x2, 0	Load Imm value 0 into register x2.
00116463	bltu x2, x1, 8	Branch to PC + 8 if x2 is less than x1 (unsigned).
00000000	nop	No operation.
00116463	bltu x2, x1, 8	Branch to PC + 8 if x2 is less than x1 (unsigned).
00000000	nop	No operation.
00116463	bltu x2, x1, 8	Branch to PC + 8 if x2 is less than x1 (unsigned).
00000000	nop	No operation.
00116463	bltu x2, x1, 8	Branch to PC + 8 if x2 is less than x1 (unsigned).
00000000	nop	No operation.
00116463	bltu x2, x1, 8	Branch to PC + 8 if x2 is less than x1 (unsigned).
00000000	nop	No operation.
00116463	bltu x2, x1, 8	Branch to PC + 8 if x2 is less than x1 (unsigned).
00000000	nop	No operation.
00116463	bltu x2, x1, 8	Branch to PC + 8 if x2 is less than x1 (unsigned).
00000000	nop	No operation.
00116463	bltu x2, x1, 8	Branch to PC + 8 if x2 is less than x1 (unsigned).
00000000	nop	No operation.
00116463	bltu x2, x1, 8	Branch to PC + 8 if x2 is less than x1 (unsigned).
00000000	nop	No operation.
0020E463	bltu x1, x2, 8	Branch to PC + 8 if x1 is less than x2 (unsigned).
0020E463	bltu x1, x2, 8	Branch to PC + 8 if x1 is less than x2 (unsigned).
0020E463	bltu x1, x2, 8	Branch to PC + 8 if x1 is less than x2 (unsigned).
0020E463	bltu x1, x2, 8	Branch to PC + 8 if x1 is less than x2 (unsigned).
0020E463	bltu x1, x2, 8	Branch to PC + 8 if x1 is less than x2 (unsigned).
0020E463	bltu x1, x2, 8	Branch to PC + 8 if x1 is less than x2 (unsigned).
0020E463	bltu x1, x2, 8	Branch to PC + 8 if x1 is less than x2 (unsigned).
0020E463	bltu x1, x2, 8	Branch to PC + 8 if x1 is less than x2 (unsigned).
0020E463	bltu x1, x2, 8	Branch to PC + 8 if x1 is less than x2 (unsigned).
0020E463	bltu x1, x2, 8	Branch to PC + 8 if x1 is less than x2 (unsigned).
00000000	nop	No operation.
f81166e3	bltu x2, x1, -116	Branch to PC - 116 if x2 is less than x1 (unsigned).

Table B.11: Firmware characterization loop for the BNE (*B-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00100093	`addi x1, x0, 1`	Add Imm 1 to x0; store result in register x1.
00000113	`li x2, 0`	Load Imm value 0 into register x2.
00209463	`bne x1, x2, 8`	Branch to PC + 8 if x1 does not equal x2.
00000000	`nop`	No operation.
00209463	`bne x1, x2, 8`	Branch to PC + 8 if x1 does not equal x2.
00000000	`nop`	No operation.
00209463	`bne x1, x2, 8`	Branch to PC + 8 if x1 does not equal x2.
00000000	`nop`	No operation.
00209463	`bne x1, x2, 8`	Branch to PC + 8 if x1 does not equal x2.
00000000	`nop`	No operation.
00209463	`bne x1, x2, 8`	Branch to PC + 8 if x1 does not equal x2.
00000000	`nop`	No operation.
00209463	`bne x1, x2, 8`	Branch to PC + 8 if x1 does not equal x2.
00000000	`nop`	No operation.
00209463	`bne x1, x2, 8`	Branch to PC + 8 if x1 does not equal x2.
00000000	`nop`	No operation.
00209463	`bne x1, x2, 8`	Branch to PC + 8 if x1 does not equal x2.
00000000	`nop`	No operation.
00209463	`bne x1, x2, 8`	Branch to PC + 8 if x1 does not equal x2.
00000000	`nop`	No operation.
00109463	`bne x1, x1, 8`	Branch to PC + 8 if x1 does not equal x1.
00109463	`bne x1, x1, 8`	Branch to PC + 8 if x1 does not equal x1.
00109463	`bne x1, x1, 8`	Branch to PC + 8 if x1 does not equal x1.
00109463	`bne x1, x1, 8`	Branch to PC + 8 if x1 does not equal x1.
00109463	`bne x1, x1, 8`	Branch to PC + 8 if x1 does not equal x1.
00109463	`bne x1, x1, 8`	Branch to PC + 8 if x1 does not equal x1.
00109463	`bne x1, x1, 8`	Branch to PC + 8 if x1 does not equal x1.
00109463	`bne x1, x1, 8`	Branch to PC + 8 if x1 does not equal x1.
00109463	`bne x1, x1, 8`	Branch to PC + 8 if x1 does not equal x1.
00109463	`bne x1, x1, 8`	Branch to PC + 8 if x1 does not equal x1.
00000000	`nop`	No operation.
F81116E3	`bne x2, x1, -116`	Branch to PC - 116 if x2 does not equal x1.

Table B.12: Firmware characterization loop for the *JAL* (*J-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
0240006F	jal x0, 36	Unconditional relative jump forward to PC + 36.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
0280006F	jal x0, 40	Unconditional relative jump forward to PC + 40.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
0280006F	jal x0, 40	Unconditional relative jump forward to PC + 40.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
00000000	nop	No operation.
F8DFF06F	jal x0, -116	Unconditional relative jump backward to PC - 116.
00000000	nop	No operation.
00000000	nop	No operation.

Table B.13: Firmware characterization loop for the *Loop-Firmware* instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
07C0006F	`jal x0, 124`	Unconditional relative jump forward to PC + 124.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
f85ff06F	`jal x0, -124`	Unconditional relative jump backward to PC - 124.

Table B.14: Firmware characterization loop for the $JALR$ (J-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
02400067	`jalr x0, x0, 36`	Jump and link register; PC = x0 + 36.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
04800067	`jalr x0, x0, 72`	Jump and link register; PC = x0 + 72.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
06C00067	`jalr x0, x0, 108`	Jump and link register; PC = x0 + 108.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
07C00067	`jalr x0, x0, 124`	Jump and link register; PC = x0 + 124.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000000	`nop`	No operation.
00000067	`jalr x0, x0, 0`	Jump and link register; PC = x0 + 0.

Table B.15: Firmware characterization loop for the LB (I-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00000297	auipc x5, 0x00000	Add 0x00000000 to PC; store address in register x5.
00028293	addi x5, x5, 40	Add Imm 40 to x5; store result in register x5.
CAFEB537	lui x10, 0xCAFEB	Load upper Imm 0xCAFEB000 into register x10.
DEADB5B7	lui x11, 0xDEADB	Load upper Imm 0xDEADB000 into register x11.
00A2A023	sb x10, 0(x5)	Store 8-bit value from x10 into memory at x5 + 0.
00B2A223	sb x11, 4(x5)	Store 8-bit value from x11 into memory at x5 + 4.
00028503	lb x10, 0(x5)	Load sign-extended 8-bit value from x5 + 0 into x10.
00428583	lb x11, 4(x5)	Load sign-extended 8-bit value from x5 + 4 into x11.
00028603	lb x12, 0(x5)	Load sign-extended 8-bit value from x5 + 0 into x12.
00428683	lb x13, 4(x5)	Load sign-extended 8-bit value from x5 + 4 into x13.
00028703	lb x14, 0(x5)	Load sign-extended 8-bit value from x5 + 0 into x14.
00428783	lb x15, 4(x5)	Load sign-extended 8-bit value from x5 + 4 into x15.
00028803	lb x16, 0(x5)	Load sign-extended 8-bit value from x5 + 0 into x16.
00428883	lb x17, 4(x5)	Load sign-extended 8-bit value from x5 + 4 into x17.
00028903	lb x18, 0(x5)	Load sign-extended 8-bit value from x5 + 0 into x18.
00428983	lb x19, 4(x5)	Load sign-extended 8-bit value from x5 + 4 into x19.
00028A03	lb x20, 0(x5)	Load sign-extended 8-bit value from x5 + 0 into x20.
00428A83	lb x21, 4(x5)	Load sign-extended 8-bit value from x5 + 4 into x21.
00028B03	lb x22, 0(x5)	Load sign-extended 8-bit value from x5 + 0 into x22.
00428B83	lb x23, 4(x5)	Load sign-extended 8-bit value from x5 + 4 into x23.
00028C03	lb x24, 0(x5)	Load sign-extended 8-bit value from x5 + 0 into x24.
00428C83	lb x25, 4(x5)	Load sign-extended 8-bit value from x5 + 4 into x25.
00028D03	lb x26, 0(x5)	Load sign-extended 8-bit value from x5 + 0 into x26.
00428D83	lb x27, 4(x5)	Load sign-extended 8-bit value from x5 + 4 into x27.
00028E03	lb x28, 0(x5)	Load sign-extended 8-bit value from x5 + 0 into x28.
00428E83	lb x29, 4(x5)	Load sign-extended 8-bit value from x5 + 4 into x29.
00028F03	lb x30, 0(x5)	Load sign-extended 8-bit value from x5 + 0 into x30.
00428F83	lb x31, 4(x5)	Load sign-extended 8-bit value from x5 + 4 into x31.
00428E83	lb x29, 4(x5)	Load sign-extended 8-bit value from x5 + 4 into x29.
03908093	addi x1, x1, 57	Add Imm 57 to x1; store result in register x1.
0012A023	sb x1, 0(x5)	Store 8-bit value from x1 into memory at x5 + 0.
F9DFF06F	jal x0, -96	Unconditional relative jump backward to PC - 96.

Table B.16: Firmware characterization loop for the LBU (I-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00000297	auipc x5, 0x00000	Add 0x00000000 to PC; store address in register x5.
00028293	addi x5, x5, 40	Add Imm 40 to x5; store result in register x5.
CAFEB537	lui x10, 0xCAFEB	Load upper Imm 0xCAFEB000 into register x10.
DEADB5B7	lui x11, 0xDEADB	Load upper Imm 0xDEADB000 into register x11.
00A2A023	sb x10, 0(x5)	Store 8-bit value from x10 into memory at x5 + 0.
00B2A223	sb x11, 4(x5)	Store 8-bit value from x11 into memory at x5 + 4.
0002C503	lbu x10, 0(x5)	Load zero-extended 8-bit value from x5 + 0 into x10.
0042C583	lbu x11, 4(x5)	Load zero-extended 8-bit value from x5 + 4 into x11.
0002C603	lbu x12, 0(x5)	Load zero-extended 8-bit value from x5 + 0 into x12.
0042C683	lbu x13, 4(x5)	Load zero-extended 8-bit value from x5 + 4 into x13.
0002C703	lbu x14, 0(x5)	Load zero-extended 8-bit value from x5 + 0 into x14.
0042C783	lbu x15, 4(x5)	Load zero-extended 8-bit value from x5 + 4 into x15.
0002C803	lbu x16, 0(x5)	Load zero-extended 8-bit value from x5 + 0 into x16.
0042C883	lbu x17, 4(x5)	Load zero-extended 8-bit value from x5 + 4 into x17.
0002C903	lbu x18, 0(x5)	Load zero-extended 16-bit value from x5 + 0 into x18.
0042C983	lbu x19, 4(x5)	Load zero-extended 16-bit value from x5 + 4 into x19.
0002CA03	lbu x20, 0(x5)	Load zero-extended 8-bit value from x5 + 0 into x20.
0042CA83	lbu x21, 4(x5)	Load zero-extended 8-bit value from x5 + 4 into x21.
0002CB03	lbu x22, 0(x5)	Load zero-extended 8-bit value from x5 + 0 into x22.
0042CB83	lbu x23, 4(x5)	Load zero-extended 8-bit value from x5 + 4 into x23.
0002CC03	lbu x24, 0(x5)	Load zero-extended 8-bit value from x5 + 0 into x24.
0042CC83	lbu x25, 4(x5)	Load zero-extended 8-bit value from x5 + 4 into x25.
0002CD03	lbu x26, 0(x5)	Load zero-extended 8-bit value from x5 + 0 into x26.
0042CD83	lbu x27, 4(x5)	Load zero-extended 8-bit value from x5 + 4 into x27.
0002CE03	lbu x28, 0(x5)	Load zero-extended 8-bit value from x5 + 0 into x28.
0042CE83	lbu x29, 4(x5)	Load zero-extended 8-bit value from x5 + 4 into x29.
0002CF03	lbu x30, 0(x5)	Load zero-extended 8-bit value from x5 + 0 into x30.
0042CF83	lbu x31, 4(x5)	Load zero-extended 8-bit value from x5 + 4 into x31.
0042CE83	lbu x29, 4(x5)	Load zero-extended 8-bit value from x5 + 4 into x29.
03908093	addi x1, x1, 57	Add Imm 57 to x1; store result in register x1.
0012A023	sb x1, 0(x5)	Store 8-bit value from x1 into memory at x5 + 0.
F9DFF06F	jal x0, -96	Unconditional relative jump backward to PC - 96.

Table B.17: Firmware characterization loop for the LH (I-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00000297	auipc x5, 0x00000	Add 0x00000000 to PC; store address in register x5.
00028293	addi x5, x5, 40	Add Imm 40 to x5; store result in register x5.
CAFEB537	lui x10, 0xCAFEB	Load upper Imm 0xCAFEB000 into register x10.
DEADB5B7	lui x11, 0xDEADB	Load upper Imm 0xDEADB000 into register x11.
00A2A023	sb x10, 0(x5)	Store 8-bit value from x10 into memory at x5 + 0.
00B2A223	sb x11, 4(x5)	Store 8-bit value from x11 into memory at x5 + 4.
00029503	lh x10, 0(x5)	Load sign-extended 16-bit value from x5 + 0 into x10.
00429583	lh x11, 4(x5)	Load sign-extended 16-bit value from x5 + 4 into x11.
00029603	lh x12, 0(x5)	Load sign-extended 16-bit value from x5 + 0 into x12.
00429683	lh x13, 4(x5)	Load sign-extended 16-bit value from x5 + 4 into x13.
00029703	lh x14, 0(x5)	Load sign-extended 16-bit value from x5 + 0 into x14.
00429783	lh x15, 4(x5)	Load sign-extended 16-bit value from x5 + 4 into x15.
00029803	lh x16, 0(x5)	Load sign-extended 16-bit value from x5 + 0 into x16.
00429883	lh x17, 4(x5)	Load sign-extended 16-bit value from x5 + 4 into x17.
00029903	lh x18, 0(x5)	Load sign-extended 16-bit value from x5 + 0 into x18.
00429983	lh x19, 4(x5)	Load sign-extended 16-bit value from x5 + 4 into x19.
00029A03	lh x20, 0(x5)	Load sign-extended 16-bit value from x5 + 0 into x20.
00429A83	lh x21, 4(x5)	Load sign-extended 16-bit value from x5 + 4 into x21.
00029B03	lh x22, 0(x5)	Load sign-extended 16-bit value from x5 + 0 into x22.
00429B83	lh x23, 4(x5)	Load sign-extended 16-bit value from x5 + 4 into x23.
00029C03	lh x24, 0(x5)	Load sign-extended 16-bit value from x5 + 0 into x24.
00429C83	lh x25, 4(x5)	Load sign-extended 16-bit value from x5 + 4 into x25.
00029D03	lh x26, 0(x5)	Load sign-extended 16-bit value from x5 + 0 into x26.
00429D83	lh x27, 4(x5)	Load sign-extended 16-bit value from x5 + 4 into x27.
00029E03	lh x28, 0(x5)	Load sign-extended 16-bit value from x5 + 0 into x28.
00429E83	lh x29, 4(x5)	Load sign-extended 16-bit value from x5 + 4 into x29.
00029F03	lh x30, 0(x5)	Load sign-extended 16-bit value from x5 + 0 into x30.
00429F83	lh x31, 4(x5)	Load sign-extended 16-bit value from x5 + 4 into x31.
00429E83	lh x29, 4(x5)	Load sign-extended 16-bit value from x5 + 4 into x29.
03908093	addi x1, x1, 57	Add Imm 57 to x1; store result in register x1.
0012A023	sb x1, 0(x5)	Store 8-bit value from x1 into memory at x5 + 0.
F9DFF06F	jal x0, -96	Unconditional relative jump backward to PC - 96.

Table B.18: Firmware characterization loop for the LHU (I-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00000297	auipc x5, 0x00000	Add 0x00000000 to PC; store address in register x5.
00028293	addi x5, x5, 40	Add Imm 40 to x5; store result in register x5.
CAFEB537	lui x10, 0xCAFEB	Load upper Imm 0xCAFEB000 into register x10.
DEADB5B7	lui x11, 0xDEADB	Load upper Imm 0xDEADB000 into register x11.
00A2A023	sb x10, 0(x5)	Store 8-bit value from x10 into memory at x5 + 0.
00B2A223	sb x11, 4(x5)	Store 8-bit value from x11 into memory at x5 + 4.
0002D503	lhu x10, 0(x5)	Load zero-extended 16-bit value from x5 + 0 into x10.
0042D583	lhu x11, 4(x5)	Load zero-extended 16-bit value from x5 + 4 into x11.
0002D603	lhu x12, 0(x5)	Load zero-extended 16-bit value from x5 + 0 into x12.
0042D683	lhu x13, 4(x5)	Load zero-extended 16-bit value from x5 + 4 into x13.
0002D703	lhu x14, 0(x5)	Load zero-extended 16-bit value from x5 + 0 into x14.
0042D783	lhu x15, 4(x5)	Load zero-extended 16-bit value from x5 + 4 into x15.
0002D803	lhu x16, 0(x5)	Load zero-extended 16-bit value from x5 + 0 into x16.
0042D883	lhu x17, 4(x5)	Load zero-extended 16-bit value from x5 + 4 into x17.
0002D903	lhu x18, 0(x5)	Load zero-extended 16-bit value from x5 + 0 into x18.
0042D983	lhu x19, 4(x5)	Load zero-extended 16-bit value from x5 + 4 into x19.
0002DA03	lhu x20, 0(x5)	Load zero-extended 16-bit value from x5 + 0 into x20.
0042DA83	lhu x21, 4(x5)	Load zero-extended 16-bit value from x5 + 4 into x21.
0002DB03	lhu x22, 0(x5)	Load zero-extended 16-bit value from x5 + 0 into x22.
0042DB83	lhu x23, 4(x5)	Load zero-extended 16-bit value from x5 + 4 into x23.
0002DC03	lhu x24, 0(x5)	Load zero-extended 16-bit value from x5 + 0 into x24.
0042DC83	lhu x25, 4(x5)	Load zero-extended 16-bit value from x5 + 4 into x25.
0002DD03	lhu x26, 0(x5)	Load zero-extended 16-bit value from x5 + 0 into x26.
0042DD83	lhu x27, 4(x5)	Load zero-extended 16-bit value from x5 + 4 into x27.
0002DE03	lhu x28, 0(x5)	Load zero-extended 16-bit value from x5 + 0 into x28.
0042DE83	lhu x29, 4(x5)	Load zero-extended 16-bit value from x5 + 4 into x29.
0002DF03	lhu x30, 0(x5)	Load zero-extended 16-bit value from x5 + 0 into x30
0042DF83	lhu x31, 4(x5)	Load zero-extended 16-bit value from x5 + 4 into x31.
0042DE83	lhu x29, 4(x5)	Load zero-extended 16-bit value from x5 + 4 into x29.
03908093	addi x1, x1, 57	Add Imm 57 to x1; store result in register x1.
0012A023	sb x1, 0(x5)	Store 8-bit value from x1 into memory at x5 + 0.
F9DFF06F	jal x0, -96	Unconditional relative jump backward to PC - 96.

Table B.19: Firmware characterization loop for the *LUI* (*U-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
AAAAA537	lui x10, 0xAAAAA	Load upper Imm 0xAAAAA000 into register x10.
555555B7	lui x11, 0x55555	Load upper Imm 0x55555000 into register x11.
F0F0F637	lui x12, 0xF0F0F	Load upper Imm 0xF0F0F000 into register x12.
0F0F06B7	lui x13, 0x0F0F0	Load upper Imm 0x0F0F0000 into register x13.
CCCC0737	lui x14, 0xCCCC0	Load upper Imm 0xCCCC0000 into register x14.
333307B7	lui x15, 0x33330	Load upper Imm 0x33330000 into register x15.
FFFF0837	lui x16, 0xFFFF0	Load upper Imm 0xFFFF0000 into register x16.
000008B7	lui x17, 0x00000	Load upper Imm 0x00000000 into register x17.
A5A5A937	lui x18, 0xA5A5A	Load upper Imm 0xA5A5A000 into register x18.
5A5A59B7	lui x19, 0x5A5A5	Load upper Imm 0x5A5A5000 into register x19.
55555537	lui x10, 0x55555	Load upper Imm 0x55555000 into register x10.
AAAAA5B7	lui x11, 0xAAAAA	Load upper Imm 0xAAAAA000 into register x11.
0F0F0637	lui x12, 0x0F0F0	Load upper Imm 0x0F0F0000 into register x12.
F0F0F6B7	lui x13, 0xF0F0F	Load upper Imm 0xF0F0F000 into register x13.
33330737	lui x14, 0x33330	Load upper Imm 0x33330000 into register x14.
CCCC07B7	lui x15, 0xCCCC0	Load upper Imm 0xCCCC0000 into register x15.
00000837	lui x16, 0x00000	Load upper Imm 0x00000000 into register x16.
FFFF08B7	lui x17, 0xFFFF0	Load upper Imm 0xFFFF0000 into register x17.
5A5A5937	lui x18, 0x5A5A5	Load upper Imm 0x5A5A5000 into register x18.
A5A5A9B7	lui x19, 0xA5A5A	Load upper Imm 0xA5A5A000 into register x19.
F0F0F537	lui x10, 0xF0F0F	Load upper Imm 0xF0F0F000 into register x10.
0F0F05B7	lui x11, 0x0F0F0	Load upper Imm 0x0F0F0000 into register x11.
33330637	lui x12, 0x33330	Load upper Imm 0x33330000 into register x12.
CCCC06B7	lui x13, 0xCCCC0	Load upper Imm 0xCCCC0000 into register x13.
00000737	lui x14, 0x00000	Load upper Imm 0x00000000 into register x14.
FFFF07B7	lui x15, 0xFFFF0	Load upper Imm 0xFFFF0000 into register x15.
5A5A5837	lui x16, 0x5A5A5	Load upper Imm 0x5A5A5000 into register x16.
A5A5A8B7	lui x17, 0xA5A5A	Load upper Imm 0xA5A5A000 into register x17.
AAAAA937	lui x18, 0xAAAAA	Load upper Imm 0xAAAAA000 into register x18.
555559B7	lui x19, 0x55555	Load upper Imm 0x55555000 into register x19.
5A5A5837	lui x16, 0x5A5A5	Load upper Imm 0x5A5A5000 into register x16.
F85FF06F	jal x0, -124	Unconditional relative jump backward to PC - 124.

Table B.20: Firmware characterization loop for the *LW* (*I-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00000297	auipc x5, 0	Add 0 to PC shifted left by 12 bits; store address in x5.
00028293	addi x5, x5, 0	Copy contents of register x5 into register x5.
CAFEB537	lui x10, 0xCAFEB	Load 20-bit Imm 0xCAFEB into upper bits of x10.
DEADB5B7	lui x11, 0xDEADB	Load 20-bit Imm 0xDEADB into upper bits of x11.
00A2A023	sw x10, 0(x5)	Store 32-bit word from x10 to memory M[x5 + 0].
00B2A223	sw x11, 4(x5)	Store 32-bit word from x11 to memory M[x5 + 4].
0002A503	lw x10, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x10.
0042A583	lw x11, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x11.
0002A603	lw x12, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x12.
0042A683	lw x13, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x13.
0002A703	lw x14, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x14.
0042A783	lw x15, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x15.
0002A803	lw x16, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x16.
0042A883	lw x17, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x17.
0002A903	lw x18, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x18.
0042A983	lw x19, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x19.
0002AA03	lw x20, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x20.
0042AA83	lw x21, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x21.
0002AB03	lw x22, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x22.
0042AB83	lw x23, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x23.
0002AC03	lw x24, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x24.
0042AC83	lw x25, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x25.
0002AD03	lw x26, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x26.
0042AD83	lw x27, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x27.
0002AE03	lw x28, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x28.
0042AE83	lw x29, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x29.
0002AF03	lw x30, 0(x5)	Load 32-bit word from memory M[x5 + 0] into x30.
0042AF83	lw x31, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x31.
0042AE83	lw x29, 4(x5)	Load 32-bit word from memory M[x5 + 4] into x29.
03908093	addi x1, x1, 57	Add Imm 57 (0x39) to register x1.
0012A023	sw x1, 0(x5)	Store 32-bit word from x1 to memory M[x5 + 0].
F9DFF06F	jal x0, -100	Unconditional relative jump to PC - 100.

Table B.21: Firmware characterization loop for the OR (R-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
37B08093	addi x1, x1, 891	Add Imm 891 to x1; store result in register x1.
7FF10113	addi x2, x2, 2047	Add Imm 2047 to x2; store result in register x2.
70ff8f93	addi x31, x31, 1807	Add Imm 1807 to x31; store result in register x31.
0020E0B3	or x1, x1, x2	Bitwise OR of x1 and x2; result in register x1.
01F0E0B3	or x1, x1, x31	Bitwise OR of x1 and x31; result in register x1.
0010E0B3	or x1, x1, x1	Bitwise OR of x1 and x1; result in register x1.
01F0EFB3	or x31, x1, x31	Bitwise OR of x1 and x31; result in register x31.
0020EFB3	or x31, x1, x2	Bitwise OR of x1 and x2; result in register x31.
0020EFB3	or x31, x1, x2	Bitwise OR of x1 and x2; result in register x31.
00116133	or x2, x2, x1	Bitwise OR of x2 and x1; result in register x2.
0020EFB3	or x31, x1, x2	Bitwise OR of x1 and x2; result in register x31.
01F0E0B3	or x1, x1, x31	Bitwise OR of x1 and x31; result in register x1.
0020E0B3	or x1, x1, x2	Bitwise OR of x1 and x2; result in register x1.
00116133	or x2, x2, x1	Bitwise OR of x2 and x1; result in register x2.
01116133	or x2, x2, x17	Bitwise OR of x2 and x17; result in register x2.
01F0EFB3	or x31, x1, x31	Bitwise OR of x1 and x31; result in register x31.
0020EFB3	or x31, x1, x2	Bitwise OR of x1 and x2; result in register x31.
0020EFB3	or x31, x1, x2	Bitwise OR of x1 and x2; result in register x31.
00116133	or x2, x2, x1	Bitwise OR of x2 and x1; result in register x2.
0020EFB3	or x31, x1, x2	Bitwise OR of x1 and x2; result in register x31.
01F0E0B3	or x1, x1, x31	Bitwise OR of x1 and x31; result in register x1.
0020E0B3	or x1, x1, x2	Bitwise OR of x1 and x2; result in register x1.
01F0EFB3	or x31, x1, x31	Bitwise OR of x1 and x31; result in register x31.
0020EFB3	or x31, x1, x2	Bitwise OR of x1 and x2; result in register x31.
0020EFB3	or x31, x1, x2	Bitwise OR of x1 and x2; result in register x31.
00116133	or x2, x2, x1	Bitwise OR of x2 and x1; result in register x2.
0020EFB3	or x31, x1, x2	Bitwise OR of x1 and x2; result in register x31.
01F0E0B3	or x1, x1, x31	Bitwise OR of x1 and x31; result in register x1.
0020E0B3	or x1, x1, x2	Bitwise OR of x1 and x2; result in register x1.
01F0EFB3	or x31, x1, x31	Bitwise OR of x1 and x31; result in register x31.
0020EFB3	or x31, x1, x2	Bitwise OR of x1 and x2; result in register x31.
F91FF06F	jal x0, -112	Unconditional relative jump backward to PC - 112.

Table B.22: Firmware characterization loop for the *ORI* (*I-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
01a0e093	ori x1, x1, 26	Bitwise OR of x1 and Imm 26; store in register x1.
2C10E113	ori x2, x1, 705	Bitwise OR of x1 and Imm 705; store in register x2.
04316193	ori x3, x2, 67	Bitwise OR of x2 and Imm 67; store in register x3.
1A41E213	ori x4, x3, 420	Bitwise OR of x3 and Imm 420; store in register x4.
08526293	ori x5, x4, 133	Bitwise OR of x4 and Imm 133; store in register x5.
3162E313	ori x6, x5, 790	Bitwise OR of x5 and Imm 790; store in register x6.
0C736393	ori x7, x6, 199	Bitwise OR of x6 and Imm 199; store in register x7.
2183E413	ori x8, x7, 536	Bitwise OR of x7 and Imm 536; store in register x8.
10946493	ori x9, x8, 265	Bitwise OR of x8 and Imm 265; store in register x9.
3A44E513	ori x10, x9, 932	Bitwise OR of x9 and Imm 932; store in register x10.
14B56593	ori x11, x10, 331	Bitwise OR of x10 and Imm 331; store in register x11.
25C5E613	ori x12, x11, 604	Bitwise OR of x11 and Imm 604; store in register x12.
18D66693	ori x13, x12, 397	Bitwise OR of x12 and Imm 397; store in register x13.
36E6E713	ori x14, x13, 878	Bitwise OR of x13 and Imm 878; store in register x14.
1CF76793	ori x15, x14, 463	Bitwise OR of x14 and Imm 463; store in register x15.
2907E813	ori x16, x15, 656	Bitwise OR of x15 and Imm 656; store in register x16.
21186893	ori x17, x16, 529	Bitwise OR of x16 and Imm 529; store in register x17.
3228E913	ori x18, x17, 802	Bitwise OR of x17 and Imm 802; store in register x18.
24396993	ori x19, x18, 579	Bitwise OR of x18 and Imm 579; store in register x19.
1549EA13	ori x20, x19, 340	Bitwise OR of x19 and Imm 340; store in register x20.
3A5A6A93	ori x21, x20, 933	Bitwise OR of x20 and Imm 933; store in register x21.
266AEB13	ori x22, x21, 614	Bitwise OR of x21 and Imm 614; store in register x22.
177B6B93	ori x23, x22, 375	Bitwise OR of x22 and Imm 375; store in register x23.
388BEC13	ori x24, x23, 904	Bitwise OR of x23 and Imm 904; store in register x24.
199C6C93	ori x25, x24, 409	Bitwise OR of x24 and Imm 409; store in register x25.
2AAAE113	ori x2, x21, 682	Bitwise OR of x21 and Imm 682; store in register x2.
1BBDE113	ori x2, x27, 443	Bitwise OR of x27 and Imm 443; store in register x2.
3CCDE113	ori x2, x27, 972	Bitwise OR of x27 and Imm 972; store in register x2.
1DDDE113	ori x2, x27, 477	Bitwise OR of x27 and Imm 477; store in register x2.
2EEAE113	ori x2, x21, 750	Bitwise OR of x21 and Imm 750; store in register x2.
1FF66F93	ori x31, x12, 511	Bitwise OR of x12 and Imm 511; store in register x31.
F85FF06F	jal x0, -124	Unconditional relative jump backward to PC - 124.

Table B.23: Firmware characterization loop for the SB (S-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00000297	auipc x5, 0x00000	Add 0x00000000 to PC; store address in register x5.
03908093	addi x1, x1, 57	Add Imm 57 to x1; store result in register x1.
00128023	sb x1, 0(x5)	Store 8-bit value from x1 into memory at x5 + 0.
00128223	sb x1, 4(x5)	Store 8-bit value from x1 into memory at x5 + 4.
00128423	sb x1, 8(x5)	Store 8-bit value from x1 into memory at x5 + 8.
00128623	sb x1, 12(x5)	Store 8-bit value from x1 into memory at x5 + 12.
00128823	sb x1, 16(x5)	Store 8-bit value from x1 into memory at x5 + 16.
00128A23	sb x1, 20(x5)	Store 8-bit value from x1 into memory at x5 + 20.
00128C23	sb x1, 24(x5)	Store 8-bit value from x1 into memory at x5 + 24.
00128E23	sb x1, 28(x5)	Store 8-bit value from x1 into memory at x5 + 28.
02128023	sb x1, 32(x5)	Store 8-bit value from x1 into memory at x5 + 32.
02128223	sb x1, 36(x5)	Store 8-bit value from x1 into memory at x5 + 36.
02128423	sb x1, 40(x5)	Store 8-bit value from x1 into memory at x5 + 40.
02128623	sb x1, 44(x5)	Store 8-bit value from x1 into memory at x5 + 44.
02128823	sb x1, 48(x5)	Store 8-bit value from x1 into memory at x5 + 48.
02128A23	sb x1, 52(x5)	Store 8-bit value from x1 into memory at x5 + 52.
02128C23	sb x1, 56(x5)	Store 8-bit value from x1 into memory at x5 + 56.
02128E23	sb x1, 60(x5)	Store 8-bit value from x1 into memory at x5 + 60.
04128023	sb x1, 64(x5)	Store 8-bit value from x1 into memory at x5 + 64.
04128223	sb x1, 68(x5)	Store 8-bit value from x1 into memory at x5 + 68.
04128423	sb x1, 72(x5)	Store 8-bit value from x1 into memory at x5 + 72.
04128623	sb x1, 76(x5)	Store 8-bit value from x1 into memory at x5 + 76.
04128823	sb x1, 80(x5)	Store 8-bit value from x1 into memory at x5 + 80.
04128A23	sb x1, 84(x5)	Store 8-bit value from x1 into memory at x5 + 84.
04128C23	sb x1, 88(x5)	Store 8-bit value from x1 into memory at x5 + 88.
04128E23	sb x1, 92(x5)	Store 8-bit value from x1 into memory at x5 + 92.
06128023	sb x1, 96(x5)	Store 8-bit value from x1 into memory at x5 + 96.
06128223	sb x1, 100(x5)	Store 8-bit value from x1 into memory at x5 + 100.
06128423	sb x1, 104(x5)	Store 8-bit value from x1 into memory at x5 + 104.
06128623	sb x1, 108(x5)	Store 8-bit value from x1 into memory at x5 + 108.
da708093	addi x1, x1, -601	Add Imm -601 to x1; store result in register x1.
f8dff06f	jal x0, -116	Unconditional relative jump backward to PC - 116.

Table B.24: Firmware characterization loop for the *SH* (*S-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00000297	auipc x5, 0x00000	Add 0x00000000 to PC; store address in register x5.
03908093	addi x1, x1, 57	Add *Imm* 57 to x1; store result in register x1.
00129023	sh x1, 0(x5)	Store 16-bit value from x1 into memory at x5 + 0.
00129223	sh x1, 4(x5)	Store 16-bit value from x1 into memory at x5 + 4.
00129423	sh x1, 8(x5)	Store 16-bit value from x1 into memory at x5 + 8.
00129623	sh x1, 12(x5)	Store 16-bit value from x1 into memory at x5 + 12.
00129823	sh x1, 16(x5)	Store 16-bit value from x1 into memory at x5 + 16.
00129A23	sh x1, 20(x5)	Store 16-bit value from x1 into memory at x5 + 20.
00129C23	sh x1, 24(x5)	Store 16-bit value from x1 into memory at x5 + 24.
00129E23	sh x1, 28(x5)	Store 16-bit value from x1 into memory at x5 + 28.
02129023	sh x1, 32(x5)	Store 16-bit value from x1 into memory at x5 + 32.
02129223	sh x1, 36(x5)	Store 16-bit value from x1 into memory at x5 + 36.
02129423	sh x1, 40(x5)	Store 16-bit value from x1 into memory at x5 + 40.
02129623	sh x1, 44(x5)	Store 16-bit value from x1 into memory at x5 + 44.
02129823	sh x1, 48(x5)	Store 16-bit value from x1 into memory at x5 + 48.
02129A23	sh x1, 52(x5)	Store 16-bit value from x1 into memory at x5 + 52.
02129C23	sh x1, 56(x5)	Store 16-bit value from x1 into memory at x5 + 56.
02129E23	sh x1, 60(x5)	Store 16-bit value from x1 into memory at x5 + 60.
04129023	sh x1, 64(x5)	Store 16-bit value from x1 into memory at x5 + 64.
04129223	sh x1, 36(x5)	Store 16-bit value from x1 into memory at x5 + 68.
04129423	sh x1, 40(x5)	Store 16-bit value from x1 into memory at x5 + 72.
04129623	sh x1, 44(x5)	Store 16-bit value from x1 into memory at x5 + 76.
04129823	sh x1, 48(x5)	Store 16-bit value from x1 into memory at x5 + 80.
04129A23	sh x1, 52(x5)	Store 16-bit value from x1 into memory at x5 + 84.
04129C23	sh x1, 56(x5)	Store 16-bit value from x1 into memory at x5 + 88.
04129E23	sh x1, 60(x5)	Store 16-bit value from x1 into memory at x5 + 92.
06129023	sh x1, 96(x5)	Store 16-bit value from x1 into memory at x5 + 96.
06129223	sh x1, 100(x5)	Store 16-bit value from x1 into memory at x5 + 100.
06129423	sh x1, 104(x5)	Store 16-bit value from x1 into memory at x5 + 104.
06129623	sh x1, 108(x5)	Store 16-bit value from x1 into memory at x5 + 108.
da708093	addi x1, x1, -601	Add *Imm* -601 to x1; store result in register x1.
f8dff06f	jal x0, -116	Unconditional relative jump backward to PC - 116.

Table B.25: Firmware characterization loop for the SLL (R-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
37B08093	addi x1, x1, 891	Add Imm 891 to x1; store result in register x1.
7FF10113	addi x2, x2, 2047	Add Imm 2047 to x2; store result in register x2.
70ff8f93	addi x31, x31, 1807	Add Imm 1807 to x31; store result in register x31.
002090B3	sll x1, x1, x2	Shift left logical x1 by amount in x2; store in x1.
01F090B3	sll x1, x1, x31	Shift left logical x1 by amount in x31; store in x1.
001090B3	sll x1, x1, x1	Shift left logical x1 by amount in x1; store in x1.
01F09FB3	sll x31, x1, x31	Shift left logical x1 by amount in x31; store in x31.
00209FB3	sll x31, x1, x2	Shift left logical x1 by amount in x2; store in x31.
01F19FB3	sll x31, x3, x31	Shift left logical x3 by amount in x31; store in x31.
00111133	sll x2, x2, x1	Shift left logical x2 by amount in x1; store in x2.
01F11133	sll x2, x2, x31	Shift left logical x2 by amount in x31; store in x2.
01F090B3	sll x1, x1, x31	Shift left logical x1 by amount in x31; store in x1.
002090B3	sll x1, x1, x2	Shift left logical x1 by amount in x2; store in x1.
00111133	sll x2, x2, x1	Shift left logical x2 by amount in x1; store in x2.
01111133	sll x2, x2, x17	Shift left logical x2 by amount in x17; store in x2.
01F09FB3	sll x31, x1, x31	Shift left logical x1 by amount in x31; store in x31.
00209FB3	sll x31, x1, x2	Shift left logical x1 by amount in x2; store in x31.
01F19FB3	sll x31, x3, x31	Shift left logical x3 by amount in x31; store in x31.
00111133	sll x2, x2, x1	Shift left logical x2 by amount in x1; store in x2.
01F11133	sll x2, x2, x31	Shift left logical x2 by amount in x31; store in x2.
01F090B3	sll x1, x1, x31	Shift left logical x1 by amount in x31; store in x1.
002090B3	sll x1, x1, x2	Shift left logical x1 by amount in x2; store in x1.
01F09FB3	sll x31, x1, x31	Shift left logical x1 by amount in x31; store in x31.
00209FB3	sll x31, x1, x2	Shift left logical x1 by amount in x2; store in x31.
01F19FB3	sll x31, x3, x31	Shift left logical x3 by amount in x31; store in x31.
00111133	sll x2, x2, x1	Shift left logical x2 by amount in x1; store in x2.
01F11133	sll x2, x2, x31	Shift left logical x2 by amount in x31; store in x2.
01F090B3	sll x1, x1, x31	Shift left logical x1 by amount in x31; store in x1.
002090B3	sll x1, x1, x2	Shift left logical x1 by amount in x2; store in x1.
01F09FB3	sll x31, x1, x31	Shift left logical x1 by amount in x31; store in x31.
00209FB3	sll x31, x1, x2	Shift left logical x1 by amount in x2; store in x31.
F91FF06F	jal x0, -112	Unconditional relative jump backward to PC - 112.

Table B.26: Firmware characterization loop for the $SLLI$ (I-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00A11113	slli x2, x2, 10	Shift left logical x2 by Imm 10; store in register x2.
00A09093	slli x1, x1, 10	Shift left logical x1 by Imm 10; store in register x1.
00A11113	slli x2, x2, 10	Shift left logical x2 by Imm 10; store in register x2.
01F11113	slli x2, x2, 31	Shift left logical x2 by Imm 31; store in register x2.
00A11113	slli x2, x2, 10	Shift left logical x2 by Imm 10; store in register x2.
00919193	slli x3, x3, 9	Shift left logical x3 by Imm 9; store in register x3.
00A11113	slli x2, x2, 10	Shift left logical x2 by Imm 10; store in register x2.
00B09093	slli x1, x1, 11	Shift left logical x1 by Imm 11; store in register x1.
00A11113	slli x2, x2, 10	Shift left logical x2 by Imm 10; store in register x2.
00A11113	slli x2, x2, 10	Shift left logical x2 by Imm 10; store in register x2.
00809F93	slli x31, x1, 8	Shift left logical x1 by Imm 8; store in register x31.
00DF1113	slli x2, x30, 13	Shift left logical x30 by Imm 13; store in register x2.
00419F93	slli x31, x3, 4	Shift left logical x3 by Imm 4; store in register x31.
009F9093	slli x1, x31, 9	Shift left logical x31 by Imm 9; store in register x1.
01011F93	slli x31, x2, 16	Shift left logical x2 by Imm 16; store in register x31.
00AF1193	slli x3, x30, 10	Shift left logical x30 by Imm 10; store in register x3.
01D09F93	slli x31, x1, 29	Shift left logical x1 by Imm 29; store in register x31.
00CF1113	slli x2, x30, 12	Shift left logical x30 by Imm 12; store in register x2.
00919F93	slli x31, x3, 9	Shift left logical x3 by Imm 9; store in register x31.
010F9093	slli x1, x31, 16	Shift left logical x31 by Imm 16; store in register x1.
00F11F93	slli x31, x2, 15	Shift left logical x2 by Imm 15; store in register x31.
018F9193	slli x3, x31, 24	Shift left logical x31 by Imm 24; store in register x3.
00409F93	slli x31, x1, 4	Shift left logical x1 by Imm 4; store in register x31.
007F1113	slli x2, x30, 7	Shift left logical x30 by Imm 7; store in register x2.
00119F93	slli x31, x3, 1	Shift left logical x3 by Imm 1; store in register x31.
01FF9093	slli x1, x31, 31	Shift left logical x31 by Imm 31; store in register x1.
00F11F93	slli x31, x2, 15	Shift left logical x2 by Imm 15; store in register x31.
017F9193	slli x3, x31, 23	Shift left logical x31 by Imm 23; store in register x3.
00009F93	slli x31, x1, 0	Shift left logical x1 by Imm 0; store in register x31.
01EF1113	slli x2, x30, 30	Shift left logical x30 by Imm 30; store in register x2.
01219F93	slli x31, x3, 18	Shift left logical x3 by Imm 18; store in register x31.
F85FF06F	jal x0, -124	Unconditional relative jump backward to PC - 124.

Table B.27: Firmware characterization loop for the SLT (R-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
37B08093	addi x1, x1, 891	Add Imm 891 to x1; store result in register x1.
7FF10113	addi x2, x2, 2047	Add Imm 2047 to x2; store result in register x2.
70ff8f93	addi x31, x31, 1807	Add Imm 1807 to x31; store result in register x31.
0020A0B3	slt x1, x1, x2	Set x1 to 1 if signed x1 is less than x2, else 0.
01F0A0B3	slt x1, x1, x31	Set x1 to 1 if signed x1 is less than x31, else 0.
0010A0B3	slt x1, x1, x1	Set x1 to 1 if signed x1 is less than x1, else 0.
01F0AFB3	slt x31, x1, x31	Set x31 to 1 if signed x1 is less than x31, else 0.
0020AFB3	slt x31, x1, x2	Set x31 to 1 if signed x1 is less than x2, else 0.
01F1AFB3	slt x31, x3, x31	Set x31 to 1 if signed x3 is less than x31, else 0.
00112133	slt x2, x2, x1	Set x2 to 1 if signed x2 is less than x1, else 0.
01F12133	slt x2, x2, x31	Set x2 to 1 if signed x2 is less than x31, else 0.
01F0A0B3	slt x1, x1, x31	Set x1 to 1 if signed x1 is less than x31, else 0.
0020A0B3	slt x1, x1, x2	Set x1 to 1 if signed x1 is less than x2, else 0.
00112133	slt x2, x2, x1	Set x2 to 1 if signed x2 is less than x1, else 0.
01112133	slt x2, x2, x17	Set x2 to 1 if signed x2 is less than x17, else 0.
01F0AFB3	slt x31, x1, x31	Set x31 to 1 if signed x1 is less than x31, else 0.
0020AFB3	slt x31, x1, x2	Set x31 to 1 if signed x1 is less than x2, else 0.
01F1AFB3	slt x31, x3, x31	Set x31 to 1 if signed x3 is less than x31, else 0.
00112133	slt x2, x2, x1	Set x2 to 1 if signed x2 is less than x1, else 0.
01F12133	slt x2, x2, x31	Set x2 to 1 if signed x2 is less than x31, else 0.
01F0A0B3	slt x1, x1, x31	Set x1 to 1 if signed x1 is less than x31, else 0.
0020A0B3	slt x1, x1, x2	Set x1 to 1 if signed x1 is less than x2, else 0.
01F0AFB3	slt x31, x1, x31	Set x31 to 1 if signed x1 is less than x31, else 0.
0020AFB3	slt x31, x1, x2	Set x31 to 1 if signed x1 is less than x2, else 0.
01F1AFB3	slt x31, x3, x31	Set x31 to 1 if signed x3 is less than x31, else 0.
00112133	slt x2, x2, x1	Set x2 to 1 if signed x2 is less than x1, else 0.
01F12133	slt x2, x2, x31	Set x2 to 1 if signed x2 is less than x31, else 0.
01F0A0B3	slt x1, x1, x31	Set x1 to 1 if signed x1 is less than x31, else 0.
0020A0B3	slt x1, x1, x2	Set x1 to 1 if signed x1 is less than x2, else 0.
01F0AFB3	slt x31, x1, x31	Set x31 to 1 if signed x1 is less than x31, else 0.
0020AFB3	slt x31, x1, x2	Set x31 to 1 if signed x1 is less than x2, else 0.
F91FF06F	jal x0, -112	Unconditional relative jump backward to PC - 112.

Table B.28: Firmware characterization loop for the $SLTI$ (I-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00A12113	slti x2, x2, 10	Set x2 to 1 if signed x2 is less than Imm 10, else 0.
00A0A093	slti x1, x1, 10	Set x1 to 1 if signed x1 is less than Imm 10, else 0.
01DFA293	slti x5, x31, 29	Set x5 to 1 if signed x31 is less than Imm 29, else 0.
01F12113	slti x2, x2, 31	Set x2 to 1 if signed x2 is less than Imm 31, else 0.
004FA293	slti x5, x31, 4	Set x5 to 1 if signed x31 is less than Imm 4, else 0.
0091A193	slti x3, x3, 9	Set x3 to 1 if signed x3 is less than Imm 9, else 0.
01FFA293	slti x5, x31, 31	Set x5 to 1 if signed x31 is less than Imm 31, else 0.
00B0A093	slti x1, x1, 11	Set x1 to 1 if signed x1 is less than Imm 11, else 0.
005FA293	slti x5, x31, 5	Set x5 to 1 if signed x31 is less than Imm 5, else 0.
00A12113	slti x2, x2, 10	Set x2 to 1 if signed x2 is less than Imm 10, else 0.
0080AF93	slti x31, x1, 8	Set x31 to 1 if signed x1 is less than Imm 8, else 0.
00DF2113	slti x2, x30, 13	Set x2 to 1 if signed x30 is less than Imm 13, else 0.
0041AF93	slti x31, x3, 4	Set x31 to 1 if signed x3 is less than Imm 4, else 0.
009F2093	slti x1, x31, 9	Set x1 to 1 if signed x31 is less than Imm 9, else 0.
01012F93	slti x31, x2, 16	Set x31 to 1 if signed x2 is less than Imm 16, else 0.
00AF2193	slti x3, x30, 10	Set x3 to 1 if signed x30 is less than Imm 10, else 0.
01D0AF93	slti x31, x1, 29	Set x31 to 1 if signed x1 is less than Imm 29, else 0.
00CF2113	slti x2, x30, 12	Set x2 to 1 if signed x30 is less than Imm 12, else 0.
0091AF93	slti x31, x3, 9	Set x31 to 1 if signed x3 is less than Imm 9, else 0.
010F2093	slti x1, x31, 16	Set x1 to 1 if signed x31 is less than Imm 16, else 0.
00F12F93	slti x31, x2, 15	Set x31 to 1 if signed x2 is less than Imm 15, else 0.
018F2193	slti x3, x31, 24	Set x3 to 1 if signed x31 is less than Imm 24, else 0.
0040AF93	slti x31, x1, 4	Set x31 to 1 if signed x1 is less than Imm 4, else 0.
007F2113	slti x2, x30, 7	Set x2 to 1 if signed x30 is less than Imm 7, else 0.
0011AF93	slti x31, x3, 1	Set x31 to 1 if signed x3 is less than Imm 1, else 0.
01FF2093	slti x1, x31, 31	Set x1 to 1 if signed x31 is less than Imm 31, else 0.
00F12F93	slti x31, x2, 15	Set x31 to 1 if signed x2 is less than Imm 15, else 0.
00F12F93	slti x31, x2, 15	Set x31 to 1 if signed x2 is less than Imm 15, else 0.
00CF2113	slti x2, x30, 12	Set x2 to 1 if signed x30 is less than Imm 12, else 0.
00F12F93	slti x31, x2, 15	Set x31 to 1 if signed x2 is less than Imm 15, else 0.
0040AF93	slti x31, x1, 4	Set x31 to 1 if signed x1 is less than Imm 4, else 0.
F85FF06F	jal x0, -124	Unconditional relative jump backward to PC - 124.

Table B.29: Firmware characterization loop for the *SLTIU* (*I-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00A13113	sltiu x2, x2, 10	Set x2 to 1 if unsigned x2 is less than 10, else 0.
00A0B093	sltiu x1, x1, 10	Set x1 to 1 if unsigned x1 is less than 10, else 0.
01DFB393	sltiu x7, x31, 29	Set x7 to 1 if unsigned x31 is less than 29, else 0.
01F13113	sltiu x2, x2, 31	Set x2 to 1 if unsigned x2 is less than 31, else 0.
004FB393	sltiu x7, x31, 4	Set x7 to 1 if unsigned x31 is less than 4, else 0.
0091B193	sltiu x3, x3, 9	Set x3 to 1 if unsigned x3 is less than 9, else 0.
01FFB393	sltiu x7, x31, 31	Set x7 to 1 if unsigned x31 is less than 31, else 0.
00B0B093	sltiu x1, x1, 11	Set x1 to 1 if unsigned x1 is less than 11, else 0.
005FB393	sltiu x7, x31, 5	Set x7 to 1 if unsigned x31 is less than 5, else 0.
00A13113	sltiu x2, x2, 10	Set x2 to 1 if unsigned x2 is less than 10, else 0.
0080BF93	sltiu x31, x1, 8	Set x31 to 1 if unsigned x1 is less than 8, else 0.
00DF3113	sltiu x2, x30, 13	Set x2 to 1 if unsigned x30 is less than 13, else 0.
0041BF93	sltiu x31, x3, 4	Set x31 to 1 if unsigned x3 is less than 4, else 0.
009F3093	sltiu x1, x31, 9	Set x1 to 1 if unsigned x31 is less than 9, else 0.
01013F93	sltiu x31, x2, 16	Set x31 to 1 if unsigned x2 is less than 16, else 0.
00AF3193	sltiu x3, x30, 10	Set x3 to 1 if unsigned x30 is less than 10, else 0.
01D0BF93	sltiu x31, x1, 29	Set x31 to 1 if unsigned x1 is less than 29, else 0.
00BB3113	sltiu x2, x22, 11	Set x2 to 1 if unsigned x22 is less than 11, else 0.
0191BF93	sltiu x31, x3, 25	Set x31 to 1 if unsigned x3 is less than 25, else 0.
010F3093	sltiu x1, x31, 16	Set x1 to 1 if unsigned x31 is less than 16, else 0.
00F13F93	sltiu x31, x2, 15	Set x31 to 1 if unsigned x2 is less than 15, else 0.
018F3193	sltiu x3, x31, 24	Set x3 to 1 if unsigned x31 is less than 24, else 0.
0040BF93	sltiu x31, x1, 4	Set x31 to 1 if unsigned x1 is less than 4, else 0.
007F3113	sltiu x2, x30, 7	Set x2 to 1 if unsigned x30 is less than 7, else 0.
0040BF93	sltiu x31, x1, 4	Set x31 to 1 if unsigned x1 is less than 4, else 0.
01FF3093	sltiu x1, x31, 31	Set x1 to 1 if unsigned x31 is less than 31, else 0.
00F13F93	sltiu x31, x2, 15	Set x31 to 1 if unsigned x2 is less than 15, else 0.
017F3193	sltiu x3, x31, 23	Set x3 to 1 if unsigned x31 is less than 23, else 0.
0000BF93	sltiu x31, x1, 0	Set x31 to 1 if unsigned x1 is less than 0, else 0.
01EF3113	sltiu x2, x30, 30	Set x2 to 1 if unsigned x30 is less than 30, else 0.
0121BF93	sltiu x31, x3, 18	Set x31 to 1 if unsigned x3 is less than 18, else 0.
F85FF06F	jal x0, -124	Unconditional relative jump backward to PC - 124.

Table B.30: Firmware characterization loop for the $SLTU$ (R-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
37B08093	addi x1, x1, 891	Add Imm 891 to x1; store result in register x1.
7FF10113	addi x2, x2, 2047	Add Imm 2047 to x2; store result in register x2.
70ff8f93	addi x31, x31, 1807	Add Imm 1807 to x31; store result in register x31.
0020B0B3	sltu x1, x1, x2	Set x1 to 1 if unsigned x1 is less than x2, else 0.
01F0B0B3	sltu x1, x1, x31	Set x1 to 1 if unsigned x1 is less than x31, else 0.
0010B0B3	sltu x1, x1, x1	Set x1 to 1 if unsigned x1 is less than x1, else 0.
01F0BFB3	sltu x31, x1, x31	Set x31 to 1 if unsigned x1 is less than x31, else 0.
0020BFB3	sltu x31, x1, x2	Set x31 to 1 if unsigned x1 is less than x2, else 0.
01F1BFB3	sltu x31, x3, x31	Set x31 to 1 if unsigned x3 is less than x31, else 0.
00113133	sltu x2, x2, x1	Set x2 to 1 if unsigned x2 is less than x1, else 0.
01F13133	sltu x2, x2, x31	Set x2 to 1 if unsigned x2 is less than x31, else 0.
01F0B0B3	sltu x1, x1, x31	Set x1 to 1 if unsigned x1 is less than x31, else 0.
0020B0B3	sltu x1, x1, x2	Set x1 to 1 if unsigned x1 is less than x2, else 0.
00113133	sltu x2, x2, x1	Set x2 to 1 if unsigned x2 is less than x1, else 0.
01113133	sltu x2, x2, x17	Set x2 to 1 if unsigned x2 is less than x17, else 0.
01F0BFB3	sltu x31, x1, x31	Set x31 to 1 if unsigned x1 is less than x31, else 0.
0020BFB3	sltu x31, x1, x2	Set x31 to 1 if unsigned x1 is less than x2, else 0.
01F1BFB3	sltu x31, x3, x31	Set x31 to 1 if unsigned x3 is less than x31, else 0.
00113133	sltu x2, x2, x1	Set x2 to 1 if unsigned x2 is less than x1, else 0.
01F13133	sltu x2, x2, x31	Set x2 to 1 if unsigned x2 is less than x31, else 0.
01F0B0B3	sltu x1, x1, x31	Set x1 to 1 if unsigned x1 is less than x31, else 0.
0020B0B3	sltu x1, x1, x2	Set x1 to 1 if unsigned x1 is less than x2, else 0.
01F0BFB3	sltu x31, x1, x31	Set x31 to 1 if unsigned x1 is less than x31, else 0.
0020BFB3	sltu x31, x1, x2	Set x31 to 1 if unsigned x1 is less than x2, else 0.
01F1BFB3	sltu x31, x3, x31	Set x31 to 1 if unsigned x3 is less than x31, else 0.
00113133	sltu x2, x2, x1	Set x2 to 1 if unsigned x2 is less than x1, else 0.
01F13133	sltu x2, x2, x31	Set x2 to 1 if unsigned x2 is less than x31, else 0.
01F0B0B3	sltu x1, x1, x31	Set x1 to 1 if unsigned x1 is less than x31, else 0.
0020B0B3	sltu x1, x1, x2	Set x1 to 1 if unsigned x1 is less than x2, else 0.
01F0BFB3	sltu x31, x1, x31	Set x31 to 1 if unsigned x1 is less than x31, else 0.
0020BFB3	sltu x31, x1, x2	Set x31 to 1 if unsigned x1 is less than x2, else 0.
F91FF06F	jal x0, -112	Unconditional relative jump backward to PC - 112.

Table B.31: Firmware characterization loop for the SRA (R-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
37B08093	addi x1, x1, 891	Add Imm 891 to x1; store result in register x1.
7FF10113	addi x2, x2, 2047	Add Imm 2047 to x2; store result in register x2.
70ff8f93	addi x31, x31, 1807	Add Imm 1807 to x31; store result in register x31.
4020D0B3	sra x1, x1, x2	Shift right arithmetic x1 by amount in x2; in x1.
40115133	sra x2, x2, x1	Shift right arithmetic x2 by amount in x1; in x2.
4021D1B3	sra x3, x3, x2	Shift right arithmetic x3 by amount in x2; in x3.
40325233	sra x4, x4, x3	Shift right arithmetic x4 by amount in x3; in x4.
4042D2B3	sra x5, x5, x4	Shift right arithmetic x5 by amount in x4; in x5.
40535333	sra x6, x6, x5	Shift right arithmetic x6 by amount in x5; in x6.
4063D3B3	sra x7, x7, x6	Shift right arithmetic x7 by amount in x6; in x7.
40745433	sra x8, x8, x7	Shift right arithmetic x8 by amount in x7; in x8.
4084D4B3	sra x9, x9, x8	Shift right arithmetic x9 by amount in x8; in x9.
40955533	sra x10, x10, x9	Shift right arithmetic x10 by amount in x9; in x10.
40A5D5B3	sra x11, x11, x10	Shift right arithmetic x11 by amount in x10; in x11.
40B65633	sra x12, x12, x11	Shift right arithmetic x12 by amount in x11; in x12.
40C6D6B3	sra x13, x13, x12	Shift right arithmetic x13 by amount in x12; in x13.
40D75733	sra x14, x14, x13	Shift right arithmetic x14 by amount in x13; in x14.
40E7D7B3	sra x15, x15, x14	Shift right arithmetic x15 by amount in x14; in x15.
40F85833	sra x16, x16, x15	Shift right arithmetic x16 by amount in x15; in x16.
4108D8B3	sra x17, x17, x16	Shift right arithmetic x17 by amount in x16; in x17.
41195933	sra x18, x18, x17	Shift right arithmetic x18 by amount in x17; in x18.
4129D9B3	sra x19, x19, x18	Shift right arithmetic x19 by amount in x18; in x19.
413A5A33	sra x20, x20, x19	Shift right arithmetic x20 by amount in x19; in x20.
414ADAB3	sra x21, x21, x20	Shift right arithmetic x21 by amount in x20; in x21.
415B5B33	sra x22, x22, x21	Shift right arithmetic x22 by amount in x21; in x22.
416BDBB3	sra x23, x23, x22	Shift right arithmetic x23 by amount in x22; in x23.
417C5C33	sra x24, x24, x23	Shift right arithmetic x24 by amount in x23; in x24.
418CDCB3	sra x25, x25, x24	Shift right arithmetic x25 by amount in x24; in x25.
419D5D33	sra x26, x26, x25	Shift right arithmetic x26 by amount in x25; in x26.
41ADDEB3	sra x27, x27, x26	Shift right arithmetic x27 by amount in x26; in x27.
41BE5F33	sub x30, x28, x27	Subtract x27 from x28; store result in register x30.
F91FF06F	jal x0, -112	Unconditional relative jump backward to PC - 112.

Table B.32: Firmware characterization loop for the $SRAI$ (I-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
40A15113	srai x2, x2, 10	Shift right arithmetic x2 by Imm 10; in register x2.
40A0D093	srai x1, x1, 10	Shift right arithmetic x1 by Imm 10; in register x1.
41DFD293	srai x5, x31, 29	Shift right arithmetic x31 by Imm 29; in register x5.
41F15113	srai x2, x2, 31	Shift right arithmetic x2 by Imm 31; in register x2.
41DFD293	srai x5, x31, 29	Shift right arithmetic x31 by Imm 29; in register x5.
4091D193	srai x3, x3, 9	Shift right arithmetic x3 by Imm 9; in register x3.
41DFD293	srai x5, x31, 29	Shift right arithmetic x31 by Imm 29; in register x5.
40B0D093	srai x1, x1, 11	Shift right arithmetic x1 by Imm 11; in register x1.
41DFD293	srai x5, x31, 29	Shift right arithmetic x31 by Imm 29; in register x5.
40A15113	srai x2, x2, 10	Shift right arithmetic x2 by Imm 10; in register x2.
4080DF93	srai x31, x1, 8	Shift right arithmetic x1 by Imm 8; in register x31.
40DF5113	srai x2, x30, 13	Shift right arithmetic x30 by Imm 13; in register x2.
4041DF93	srai x31, x3, 4	Shift right arithmetic x3 by Imm 4; in register x31.
409F5093	srai x1, x31, 9	Shift right arithmetic x31 by Imm 9; in register x1.
41015F93	srai x31, x2, 16	Shift right arithmetic x2 by Imm 16; in register x31.
40AF5193	srai x3, x30, 10	Shift right arithmetic x30 by Imm 10; in register x3.
41D0DF93	srai x31, x1, 29	Shift right arithmetic x1 by Imm 29; in register x31.
40CF5113	srai x2, x30, 12	Shift right arithmetic x30 by Imm 12; in register x2.
4091DF93	srai x31, x3, 9	Shift right arithmetic x3 by Imm 9; in register x31.
410F5093	srai x1, x31, 16	Shift right arithmetic x31 by Imm 16; in register x1.
40F15F93	srai x31, x2, 15	Shift right arithmetic x2 by Imm 15; in register x31.
418F5193	srai x3, x31, 24	Shift right arithmetic x31 by Imm 24; in register x3.
4040DF93	srai x31, x1, 4	Shift right arithmetic x1 by Imm 4; in register x31.
407F5113	srai x2, x30, 7	Shift right arithmetic x30 by Imm 7; in register x2.
4011DF93	srai x31, x3, 1	Shift right arithmetic x3 by Imm 1; in register x31.
41FF5093	srai x1, x31, 31	Shift right arithmetic x31 by Imm 31; in register x1.
40F15F93	srai x31, x2, 15	Shift right arithmetic x2 by Imm 15; in register x31.
417F5193	srai x3, x31, 23	Shift right arithmetic x31 by Imm 23; in register x3.
4000DF93	srai x31, x1, 0	Shift right arithmetic x1 by Imm 0; in register x31.
41EF5113	srai x2, x30, 30	Shift right arithmetic x30 by Imm 30; in register x2.
4121DF93	srai x31, x3, 18	Shift right arithmetic x3 by Imm 18; in register x31.
F85FF06F	jal x0, -124	Unconditional relative jump backward to PC - 124.

Table B.33: Firmware characterization loop for the SRL (R-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
37B08093	addi x1, x1, 891	Add Imm 891 to x1; store result in register x1.
7FF10113	addi x2, x2, 2047	Add Imm 2047 to x2; store result in register x2.
70ff8f93	addi x31, x31, 1807	Add Imm 1807 to x31; store result in register x31.
0020D0B3	srl x1, x1, x2	Shift right logical x1 by amount in x2; store in x1.
01F0D0B3	srl x1, x1, x31	Shift right logical x1 by amount in x31; store in x1.
0010D0B3	srl x1, x1, x1	Shift right logical x1 by amount in x1; store in x1.
01F0DFB3	srl x31, x1, x31	Shift right logical x1 by amount in x31; store in x31.
0020DFB3	srl x31, x1, x2	Shift right logical x1 by amount in x2; store in x31.
01F1DFB3	srl x31, x3, x31	Shift right logical x3 by amount in x31; store in x31.
00115133	srl x2, x2, x1	Shift right logical x2 by amount in x1; store in x2.
01F15133	srl x2, x2, x31	Shift right logical x2 by amount in x31; store in x2.
01F0D0B3	srl x1, x1, x31	Shift right logical x1 by amount in x31; store in x1.
0020D0B3	srl x1, x1, x2	Shift right logical x1 by amount in x2; store in x1.
00115133	srl x2, x2, x1	Shift right logical x2 by amount in x1; store in x2.
01115133	srl x2, x2, x17	Shift right logical x2 by amount in x17; store in x2.
01F0DFB3	srl x31, x1, x31	Shift right logical x1 by amount in x31; store in x31.
0020DFB3	srl x31, x1, x2	Shift right logical x1 by amount in x2; store in x31.
01F1DFB3	srl x31, x3, x31	Shift right logical x3 by amount in x31; store in x31.
00115133	srl x2, x2, x1	Shift right logical x2 by amount in x1; store in x2.
01F15133	srl x2, x2, x31	Shift right logical x2 by amount in x31; store in x2.
01F0D0B3	srl x1, x1, x31	Shift right logical x1 by amount in x31; store in x1.
0020D0B3	srl x1, x1, x2	Shift right logical x1 by amount in x2; store in x1.
01F0DFB3	srl x31, x1, x31	Shift right logical x1 by amount in x31; store in x31.
0020DFB3	srl x31, x1, x2	Shift right logical x1 by amount in x2; store in x31.
01F1DFB3	srl x31, x3, x31	Shift right logical x3 by amount in x31; store in x31.
00115133	srl x2, x2, x1	Shift right logical x2 by amount in x1; store in x2.
01F15133	srl x2, x2, x31	Shift right logical x2 by amount in x31; store in x2.
01F0D0B3	srl x1, x1, x31	Shift right logical x1 by amount in x31; store in x1.
0020D0B3	srl x1, x1, x2	Shift right logical x1 by amount in x2; store in x1.
01F0DFB3	srl x31, x1, x31	Shift right logical x1 by amount in x31; store in x31.
0020DFB3	srl x31, x1, x2	Shift right logical x1 by amount in x2; store in x31.
F91FF06F	jal x0, -112	Unconditional relative jump backward to PC - 112.

Table B.34: Firmware characterization loop for the $SRLI$ (I-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
005FD113	srli x2, x31, 5	Shift right logical x31 by Imm 5; in register x2.
00A0D093	srli x1, x1, 10	Shift right logical x1 by Imm 10; in register x1.
01FDDA13	srli x10, x27, 509	Shift right logical x10 by Imm 509; in register x27.
01F15113	srli x2, x2, 31	Shift right logical x2 by Imm 31; in register x2.
004FDF13	srli x30, x31, 4	Shift right logical x31 by Imm 4; in register x30.
0091D193	srli x3, x3, 9	Shift right logical x3 by Imm 9; in register x3.
01FFDF13	srli x30, x31, 31	Shift right logical x31 by Imm 31; in register x30.
00B0D093	srli x1, x1, 11	Shift right logical x1 by Imm 11; in register x1.
005FDF13	srli x30, x31, 5	Shift right logical x31 by Imm 5; in register x30.
00A15113	srli x2, x2, 10	Shift right logical x2 by Imm 10; in register x2.
0080DF13	srli x30, x1, 8	Shift right logical x1 by Imm 8; in register x30.
00DF5113	srli x2, x30, 13	Shift right logical x30 by Imm 13; in register x2.
0041DF13	srli x30, x3, 4	Shift right logical x3 by Imm 4; in register x30.
009FD093	srli x1, x31, 9	Shift right logical x31 by Imm 9; in register x1.
01015F13	srli x30, x2, 16	Shift right logical x2 by Imm 16; in register x30.
00AF5193	srli x3, x30, 10	Shift right logical x30 by Imm 10; in register x3.
01D0DF13	srli x30, x1, 29	Shift right logical x1 by Imm 29; in register x30.
00CF5113	srli x2, x30, 12	Shift right logical x30 by Imm 12; in register x2.
0191DF13	srli x30, x3, 25	Shift right logical x3 by Imm 25; in register x30.
0000DF13	srli x30, x1, 0	Shift right logical x1 by Imm 0; in register x30.
00F15F13	srli x30, x2, 15	Shift right logical x2 by Imm 15; in register x30.
0000DF13	srli x30, x1, 0	Shift right logical x1 by Imm 0; in register x30.
0040DF13	srli x30, x1, 4	Shift right logical x1 by Imm 4; in register x30.
007F5113	srli x2, x30, 7	Shift right logical x30 by Imm 7; in register x2.
0011DF13	srli x30, x3, 1	Shift right logical x3 by Imm 1; in register x30.
0000DF13	srli x30, x1, 0	Shift right logical x1 by Imm 0; in register x30.
00F15F13	srli x30, x2, 15	Shift right logical x2 by Imm 15; in register x30.
0000DF13	srli x30, x1, 0	Shift right logical x1 by Imm 0; in register x30.
0000DF13	srli x30, x1, 0	Shift right logical x1 by Imm 0; in register x30.
01EF5113	srli x2, x30, 30	Shift right logical x30 by Imm 30; in register x2.
0121DF13	srli x30, x3, 18	Shift right logical x3 by Imm 18; in register x30.
F85FF06F	jal x0, -124	Unconditional relative jump backward to PC - 124.

Table B.35: Firmware characterization loop for the SUB (R-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
37B08093	`addi x1, x1, 891`	Add Imm 891 to x1; store result in register x1.
7FF10113	`addi x2, x2, 2047`	Add Imm 2047 to x2; store result in register x2.
70F10113	`addi x2, x2, 1807`	Add Imm 1807 to x2; store result in register x2.
402080B3	`sub x1, x1, x2`	Subtract x2 from x1; store result in register x1.
40110133	`sub x2, x2, x1`	Subtract x1 from x2; store result in register x2.
402181B3	`sub x3, x3, x2`	Subtract x2 from x3; store result in register x3.
40320233	`sub x4, x4, x3`	Subtract x3 from x4; store result in register x4.
404282B3	`sub x5, x5, x4`	Subtract x4 from x5; store result in register x5.
40530333	`sub x6, x6, x5`	Subtract x5 from x6; store result in register x6.
406383B3	`sub x7, x7, x6`	Subtract x6 from x7; store result in register x7.
40740433	`sub x8, x8, x7`	Subtract x7 from x8; store result in register x8.
408484B3	`sub x9, x9, x8`	Subtract x8 from x9; store result in register x9.
40950533	`sub x10, x10, x9`	Subtract x9 from x10; store result in register x10.
40A585B3	`sub x11, x11, x10`	Subtract x10 from x11; store result in register x11.
40B60633	`sub x12, x12, x11`	Subtract x11 from x12; store result in register x12.
40C686B3	`sub x13, x13, x12`	Subtract x12 from x13; store result in register x13.
40D70733	`sub x14, x14, x13`	Subtract x13 from x14; store result in register x14.
40E787B3	`sub x15, x15, x14`	Subtract x14 from x15; store result in register x15.
40F80833	`sub x16, x16, x15`	Subtract x15 from x16; store result in register x16.
410888B3	`sub x17, x17, x16`	Subtract x16 from x17; store result in register x17.
41190933	`sub x18, x18, x17`	Subtract x17 from x18; store result in register x18.
412989B3	`sub x19, x19, x18`	Subtract x18 from x19; store result in register x19.
413A0A33	`sub x20, x20, x19`	Subtract x19 from x20; store result in register x20.
414A8AB3	`sub x21, x21, x20`	Subtract x20 from x21; store result in register x21.
415B0B33	`sub x22, x22, x21`	Subtract x21 from x22; store result in register x22.
416B8BB3	`sub x23, x23, x22`	Subtract x22 from x23; store result in register x23.
417C0C33	`sub x24, x24, x23`	Subtract x23 from x24; store result in register x24.
418C8CB3	`sub x25, x25, x24`	Subtract x24 from x25; store result in register x25.
419D0D33	`sub x26, x26, x25`	Subtract x25 from x26; store result in register x26.
41AD8EB3	`sub x27, x27, x26`	Subtract x26 from x27; store result in register x27.
41BE0F33	`sub x30, x28, x27`	Subtract x27 from x28; store result in register x30.
F91FF06F	`jal x0, -112`	Unconditional relative jump backward to PC - 112.

Table B.36: Firmware characterization loop for the *SW* (*S-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00000297	auipc x5, 0	Add 0 to PC shifted left by 12 bits; store address in x5.
03908093	addi x1, x1, 57	Add *Imm* 57 (0x39) to register x1.
0012A023	sw x1, 0(x5)	Store 32-bit word from x1 to memory M[x5 + 0].
0012A223	sw x1, 4(x5)	Store 32-bit word from x1 to memory M[x5 + 4].
0012A423	sw x1, 8(x5)	Store 32-bit word from x1 to memory M[x5 + 8].
0012A623	sw x1, 12(x5)	Store 32-bit word from x1 to memory M[x5 + 12].
0012A823	sw x1, 16(x5)	Store 32-bit word from x1 to memory M[x5 + 16].
0012AA23	sw x1, 20(x5)	Store 32-bit word from x1 to memory M[x5 + 20].
0012AC23	sw x1, 24(x5)	Store 32-bit word from x1 to memory M[x5 + 24].
0012AE23	sw x1, 28(x5)	Store 32-bit word from x1 to memory M[x5 + 28].
0212A023	sw x1, 32(x5)	Store 32-bit word from x1 to memory M[x5 + 32].
0212A223	sw x1, 36(x5)	Store 32-bit word from x1 to memory M[x5 + 36].
0212A423	sw x1, 40(x5)	Store 32-bit word from x1 to memory M[x5 + 40].
0212A623	sw x1, 44(x5)	Store 32-bit word from x1 to memory M[x5 + 44].
0212A823	sw x1, 48(x5)	Store 32-bit word from x1 to memory M[x5 + 48].
0212AA23	sw x1, 52(x5)	Store 32-bit word from x1 to memory M[x5 + 52].
0212AC23	sw x1, 56(x5)	Store 32-bit word from x1 to memory M[x5 + 56].
0212AE23	sw x1, 60(x5)	Store 32-bit word from x1 to memory M[x5 + 60].
0412A023	sw x1, 64(x5)	Store 32-bit word from x1 to memory M[x5 + 64].
0412A223	sw x1, 68(x5)	Store 32-bit word from x1 to memory M[x5 + 68].
0412A423	sw x1, 72(x5)	Store 32-bit word from x1 to memory M[x5 + 72].
0412A623	sw x1, 76(x5)	Store 32-bit word from x1 to memory M[x5 + 76].
0412A823	sw x1, 80(x5)	Store 32-bit word from x1 to memory M[x5 + 80].
0412AA23	sw x1, 84(x5)	Store 32-bit word from x1 to memory M[x5 + 84].
0412AC23	sw x1, 88(x5)	Store 32-bit word from x1 to memory M[x5 + 88].
0412AE23	sw x1, 92(x5)	Store 32-bit word from x1 to memory M[x5 + 92].
0612A023	sw x1, 96(x5)	Store 32-bit word from x1 to memory M[x5 + 96].
0612A223	sw x1, 100(x5)	Store 32-bit word from x1 to memory M[x5 + 100].
0612A423	sw x1, 104(x5)	Store 32-bit word from x1 to memory M[x5 + 104].
0612A623	sw x1, 108(x5)	Store 32-bit word from x1 to memory M[x5 + 108].
DA708093	addi x1, x1, -601	Add *Imm* -601 (0xDA7) to register x1.
F8DFF06F	jal x0, -116	Unconditional relative jump to PC - 116.

Table B.37: Firmware characterization loop for the *Blank-Firmware* instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation
00000000	nop	No operation

Table B.38: Firmware characterization loop for the XOR (R-$Type$) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
37B08093	addi x1, x1, 891	Add Imm 891 to x1; store result in register x1.
7FF10113	addi x2, x2, 2047	Add Imm 2047 to x2; store result in register x2.
70ff8f93	addi x31, x31, 1807	Add Imm 1807 to x31; store result in register x31.
0020C0B3	xor x1, x1, x2	XOR of x1 and x2; store result in register x1.
01F0C0B3	xor x1, x1, x31	XOR of x1 and x31; store result in register x1.
0010C0B3	xor x1, x1, x1	XOR of x1 and x1; store result in register x1.
01F0CFB3	xor x31, x1, x31	XOR of x1 and x31; store result in register x31.
0020CFB3	xor x31, x1, x2	XOR of x1 and x2; store result in register x31.
01F1CFB3	xor x31, x3, x31	XOR of x3 and x31; store result in register x31.
00114133	xor x2, x2, x1	XOR of x2 and x1; store result in register x2.
01F14133	xor x2, x2, x31	XOR of x2 and x31; store result in register x2.
01F0C0B3	xor x1, x1, x31	XOR of x1 and x31; store result in register x1.
0020C0B3	xor x1, x1, x2	XOR of x1 and x2; store result in register x1.
00114133	xor x2, x2, x1	XOR of x2 and x1; store result in register x2.
01114133	xor x2, x2, x17	XOR of x2 and x17; store result in register x2.
01F0CFB3	xor x31, x1, x31	XOR of x1 and x31; store result in register x31.
0020CFB3	xor x31, x1, x2	XOR of x1 and x2; store result in register x31.
01F1CFB3	xor x31, x3, x31	XOR of x3 and x31; store result in register x31.
00114133	xor x2, x2, x1	XOR of x2 and x1; store result in register x2.
01F14133	xor x2, x2, x31	XOR of x2 and x31; store result in register x2.
01F0C0B3	xor x1, x1, x31	XOR of x1 and x31; store result in register x1.
0020C0B3	xor x1, x1, x2	XOR of x1 and x2; store result in register x1.
01F0CFB3	xor x31, x1, x31	XOR of x1 and x31; store result in register x31.
0020CFB3	xor x31, x1, x2	XOR of x1 and x2; store result in register x31.
01F1CFB3	xor x31, x3, x31	XOR of x3 and x31; store result in register x31.
00114133	xor x2, x2, x1	XOR of x2 and x1; store result in register x2.
01F14133	xor x2, x2, x31	XOR of x2 and x31; store result in register x2.
01F0C0B3	xor x1, x1, x31	XOR of x1 and x31; store result in register x1.
0020C0B3	xor x1, x1, x2	XOR of x1 and x2; store result in register x1.
01F0CFB3	xor x31, x1, x31	XOR of x1 and x31; store result in register x31.
0020CFB3	xor x31, x1, x2	XOR of x1 and x2; store result in register x31.
F91FF06F	jal x0, -112	Unconditional relative jump backward to PC - 112.

Table B.39: Firmware characterization loop for the $XORI$ (*I-Type*) instruction.

HEX CODE	ASSEMBLY (RV32I)	DESCRIPTION
05F54593	`xori x11, x10, 95`	XOR of x10 and Imm 95; store in register x11.
AAA0C113	`xori x2, x1, -1366`	XOR of x1 and Imm -1366; store in register x2.
55514193	`xori x3, x2, 1365`	XOR of x2 and Imm 1365; store in register x3.
05F54593	`xori x11, x10, 95`	XOR of x10 and Imm 95; store in register x11.
0AA24293	`xori x5, x4, 170`	XOR of x4 and Imm 170; store in register x5.
1FF2C193	`xori x3, x5, 511`	XOR of x5 and Imm 511; store in register x3.
04F34393	`xori x7, x6, 79`	XOR of x6 and Imm 79; store in register x7.
0093C213	`xori x4, x7, 9`	XOR of x7 and Imm 9; store in register x4.
1FF44293	`xori x5, x8, 511`	XOR of x8 and Imm 511; store in register x5.
00B4C293	`xori x5, x9, 11`	XOR of x9 and Imm 11; store in register x5.
05F54593	`xori x11, x10, 95`	XOR of x10 and Imm 95; store in register x11.
0AA5C313	`xori x6, x11, 170`	XOR of x11 and Imm 170; store in register x6.
00864313	`xori x6, x12, 8`	XOR of x12 and Imm 8; store in register x6.
0DF6C393	`xori x7, x13, 223`	XOR of x13 and Imm 223; store in register x7.
04174393	`xori x7, x14, 65`	XOR of x14 and Imm 65; store in register x7.
0097C413	`xori x8, x15, 9`	XOR of x15 and Imm 9; store in register x8.
01084413	`xori x8, x16, 16`	XOR of x16 and Imm 16; store in register x8.
0AA8C493	`xori x9, x17, 170`	XOR of x17 and Imm 170; store in register x9.
1D094493	`xori x9, x18, 464`	XOR of x18 and Imm 464; store in register x9.
0CC9C513	`xori x10, x19, 204`	XOR of x19 and Imm 204; store in register x10.
00914513	`xori x10, x2, 9`	XOR of x2 and Imm 9; store in register x10.
10FA4613	`xori x12, x20, 271`	XOR of x20 and Imm 271; store in register x12.
00F14593	`xori x11, x2, 15`	XOR of x2 and Imm 15; store in register x11.
18FC4613	`xori x12, x24, 399`	XOR of x24 and Imm 399; store in register x12.
0404C093	`xori x1, x9, 64`	XOR of x9 and Imm 64; store in register x1.
07F54113	`xori x2, x10, 127`	XOR of x10 and Imm 127; store in register x2.
0115C113	`xori x2, x11, 17`	XOR of x11 and Imm 17; store in register x2.
1FF64193	`xori x3, x12, 511`	XOR of x12 and Imm 511; store in register x3.
0F16C193	`xori x3, x13, 241`	XOR of x13 and Imm 241; store in register x3.
17F74213	`xori x4, x14, 383`	XOR of x14 and Imm 383; store in register x4.
0007C213	`xori x4, x15, 0`	XOR of x15 and Imm 0; store in register x4.
F85FF06F	`jal x0, -124`	Unconditional relative jump backward to PC - 124.