Third-Year Dissertation Project EEE-381

School of Electronic and Electrical Engineering

May 2025

Project Title:

Investigation into the Feasibility of Integrating a Stochastic Computing Accelerator Unit with a RISC-V Processor on an FPGA Platform.

Name: David Cracknell

Reference No: 220139519

Supervisor: Neil Powell

Secondary Marker: Mathew Hobbs

Abstract

This investigation assesses the feasibility of attaching a lightweight Stochastic Computing Accelerator Unit (SCAU) to an open-source SweRVolf RISC-V core on a Nexys A7 FPGA. The 8-bit SCAU converts binary operands into 256-bit stochastic streams, executing a multiply-accumulate function across up to 100 parallel channels, before reconverting the result to binary. Benchmarking against a pipelined deterministic MAC shows that the SCAU trims dynamic power from 0.094 W to 0.003 W and cuts total on-chip power nearly in half while incurring a mean absolute error of 1.6%, a latency increase from 10 ns to 466 ns, and a higher energy per operation.

Statistical testing over 10,000 vectors confirms that errors remain within a single-digit percentage for typical inputs. A Wishbone-compliant wrapper and simulation waveform verify register-level hand-shaking with the processor, although full in-system execution is reserved for future work. The results demonstrate that stochastic arithmetic can yield order-of-magnitude power savings for error-tolerant edge-AI and sensing tasks and that RISC-V's modular fabric readily accommodates such approximate accelerators, provided bit-stream quality and routing overhead are further refined.

Individual Contributions

This project involved independently investigating the potential integration of a Stochastic Computing Accelerator Unit (SCAU) into a RISC-V processor implemented on an FPGA. The work began with an in-depth literature review to develop a strong understanding of stochastic computing principles, RISC-V architecture, and FPGA-based accelerator design. This review helped identify relevant research gaps and informed the direction of the practical exploration.

Throughout the project, I developed and tested various hardware modules using custom test benches to verify their functionality and evaluate feasibility. Although the project did not deliver a fully integrated SCAU system with a RISC-V SoC, the project successfully identified the key design and implementation challenges necessary to accomplish this task. I received guidance from Professors Neil Powell and Tiantai Deng, particularly in areas related to hardware design approaches and test strategies.

Hardware Platform and Tools Used

- **FPGA Board:** The Nexys A7 FPGA development board was selected for its flexibility, extensive available documentation, and compatibility with the RVfpga framework[1].
- **RISC-V Core:** The SweRVolf implementation of the RISC-V processor (provided by Imagination Technologies) was employed. SweRVolf incorporates the SweRV EH1 core optimized for embedded applications[2].
- **Hardware Description Language (HDL):** Verilog was chosen for hardware design and simulation due to its widespread use and compatibility with synthesis toolchains.
- **Development Environment:** Xilinx Vivado Design Suite was used for coding, simulation, synthesis, and FPGA implementation [3].
- **Software Development Platform:** PlatformIO was employed to program the RISC-V processor, providing IDE integration, debugging tools, and library management[4].

Glossary

AI	Artificial Intelligence
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
ISA	Instruction Set Architecture
LFSR	Linear Feedback Shift Register
MAC	Multiply-Accumulate Accelerator
MUX	Multiplexer
PlatformIO	Cross-platform embedded development environment on VS Studio
RISC-V	Reduced Instruction Set Computing – V
RVfpga	RISC-V FPGA framework
SC	Stochastic Computing
SCAU	Stochastic Computing Accelerator Unit
SNG	Stochastic Number Generator
SoC	System-on-Chip
Vivado	Xilinx Vivado Design Suite
LUT	Look-Up Table
FF	Flip flops
CNN	Convolutional Neural Network

Contents

1 Introduction

1.1 Background

Since the technology boom of the early 21st century, rapid advancements and the widespread use of artificial intelligence (AI) have driven significant demand for efficient and high-performance computing methods[5] . Traditional computational methods involving precise binary calculations have increasingly become a limiting factor for industries such as AI development. These conventional approaches are reaching major bottlenecks in power consumption, processing speed, and chip area, thereby constraining developers' ability to scale and optimise systems for progressively complex and data-intensive workloads.

Approximate computing paradigms, notably Stochastic Computing (SC), offer promising solutions by trading off exact calculation precision to improve energy efficiency and reduce circuit complexity [6] . Originally developed in the 1960s, SC represents numerical values through probabilistic bitstreams, enabling simple arithmetic operations using basic logic gates. This approach has the potential to improve noise resilience, power consumption, and circuit simplicity, making it particularly attractive for AI/neural network applications. The Reduced Instruction Set Computing (RISC-V) architecture is gaining considerable traction due to its open-source, extensible, and modular characteristics [7] . This flexibility facilitates the seamless integration of specialised accelerators, which are becoming more commonly used in AI training hardware due to their adaptability and capability to incorporate custom extensions tailored for AI acceleration chips.

Field-Programmable Gate Arrays (FPGAs) provide a versatile and reconfigurable platform ideally suited for rapid prototyping, implementation, and iterative evaluation of custom computing accelerators. Their flexibility allows the practical review of novel hardware designs, such as SC-based accelerators.

1.2 Motivation and Literature Review

The motivation for this research arises from the critical need to balance computational performance, energy efficiency, and hardware complexity in modern computing systems. SC, with its simplified arithmetic and resilience to noise, has emerged as a promising approach, particularly for energy-constrained, high error tolerance applications like edge AI and neural networks. Simultaneously, the modular and open-source RISC-V architecture provides a flexible platform for integrating specialised accelerators.

Previous work has made significant strides in leveraging SC for neural networks and data-intensive tasks. For instance, Sim and Lee [8] proposed an SC-based multiplication technique optimised for convolutional neural networks, achieving up to 490x more energy efficiency over the conventional methods but encountering challenges with bitstream latency and accuracy. Similarly, Schober [9] introduced a high-accuracy MAC unit that addressed issues of correlation in unary bitstreams, offering significant improvements in computation accuracy. Cruz [10] further demonstrated SC's applicability in convolutional neural networks on FPGAs, showing its potential for energy-efficient image classification. However, these works largely focus on isolated SC components or specific applications, leaving broader integration within modular, scalable systems like RISC-V as unexplored areas for potential research.

1.3 Project Aim and Objectives

This research aims to investigate the feasibility of designing a simple Stochastic Computing Accelerator Unit (SCAU) and explore the potential for integrating it with a RISC-V processor implemented on an FPGA board. By leveraging RISC-V's modular architecture and recent advancements in stochastic computing techniques, the study aims to assess how such integration might be achieved. Key performance metrics—such as computational speed, energy efficiency, resource utilisation, and accuracy—are evaluated by benchmarking the standalone SCAU design against a traditional pipelined multiply-accumulate (MAC) unit. Through this investigation, the work highlights the practical considerations and challenges of combining SC with modular processor architectures and offers insights into its potential role in developing scalable, energy-efficient hardware systems for energy-constrained applications.

1.4 Overview

This dissertation explores the use of SC as a potential energy-efficient alternative to conventional binary computation, with reduced energy consumption. It begins with a theoretical overview, outlining SC principles, historical development, and its comparative advantages and limitations relative to traditional arithmetic systems. The scope then extends to an examination of the RISC-V processor architecture, highlighting its modular design and suitability for accommodating custom accelerators.

Building on this theoretical foundation, this report presents the design and evaluation of a custom SCAU, developed for deployment on an FPGA platform. The feasibility of integrating such a unit with a RISC-V processor was also investigated, with particular attention given to potential interfacing strategies, control mechanisms, and architectural constraints.

2 Theory

Integrating SCAU into RISC-V processors on FPGA platforms provides a mechanism for exploration of approximate computing's potential in the hardware accelerator industry. SC offers advantages such as energy efficiency and fault tolerance, while RISC-V's modular, open-source architecture provides a flexible foundation for custom accelerators [7] . FPGAs, with their rapid prototyping capabilities, are ideal for evaluating such designs. This section reviews the theoretical principles underpinning these technologies, critically appraises existing literature, identifies research gaps, and establishes the context and motivation for this study. By addressing these gaps, this project evaluates the feasibility of integrating stochastic accelerators into RISC-V processors on FPGAs.

2.1 Stochastic Computing

SC is an unconventional computational method whereby numerical values are represented as proportions of '1's in a stochastic bitstream. Each bit is assigned a value of 0 or 1 with a probability proportional to the encoded value[6] . This approach was developed in the 1960s by Brian Gaines in the UK and Ted Poppelbaum in the US, with the initial aim of mimicking biological neural processes and improving efficiency in analogue-to-digital systems[11] . The key innovation in SC is encoding real numbers within the interval {0,1} as the proportion of '1's in a random sequence, generated a random number generator called the SNG [12] .

The operation of such a stochastic number generator (SNG) is illustrated in Figure 2.1. The SNG begins by generating a random binary number, R, which is then compared to a given binary input number, B. If R is greater than B, the comparator outputs 1; otherwise it outputs 0. This process generates one bit per clock cycle [6] . Over N clock cycles, the SNG produces an N-bit long stochastic bitstream representing X, where X is the probability that a random number picked from the bitstream is "1". The accuracy of the X depends on the degree of randomness in the generated bit pattern and the length of the stochastic bitstream generated [6] .

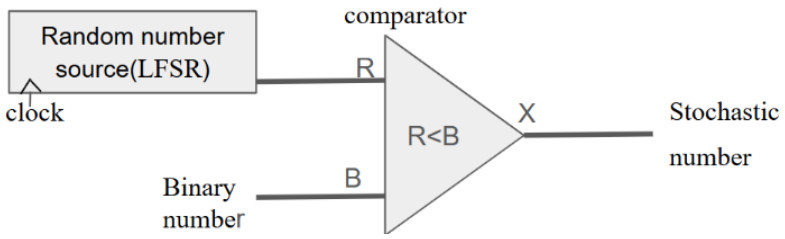

Figure 2.1: *Binary-to-Stochastic Number Generator (SNG) via LFSR-Comparator Architecture.*

This probabilistic representation enables SC circuits to tolerate noise and small errors, making SC attractive for low-cost, energy-efficient applications like neural networks [10]. However, SC fell out of favour due to its low computational precision and inefficiency for general-purpose computing tasks[6] Recently, it has seen a resurgence with the rise of AI and machine learning, where its inherent noise resilience, parallelism, and simplicity align well with the demands of approximate computing and energy-efficient hardware[8].

SC employs two primary encoding schemes:

- **Unipolar encoding.** Represents a real value $X \in [0, 1]$ as the fraction of '1's in the bitstream.
- **Bipolar encoding.** Represents $X \in [-1, 1]$ via $X = 2p - 1$, where p is the probability of a '1'.

Arithmetic operations in SC differ significantly from conventional binary systems [9, 6]:

- **Multiplication.** Unipolar uses an AND gate ($P(X \wedge Y) = P(X)P(Y)$); bipolar uses an XOR gate.
- **Addition.** Achieved via a multiplexer (weighted sum) or an OR gate ($P(X \vee Y) = P(X) + P(Y) - P(X)P(Y)$).

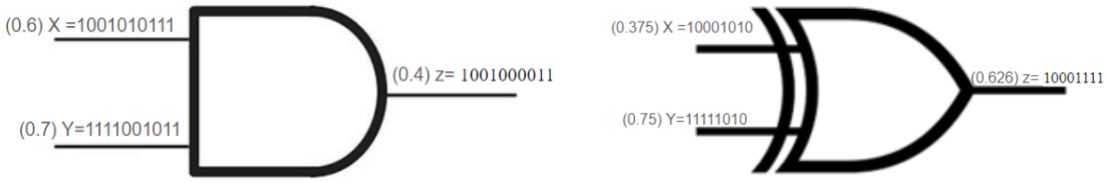

Figure 2.2: *Example SC Bitstream multiplier logic of (left) unipolar AND gate and (right) bipolar EX-OR gate*

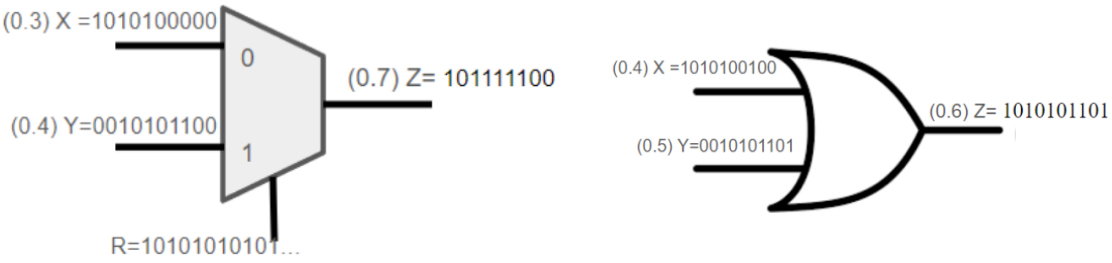

Figure 2.3: *Example of SC Bitstream addition logic of (left) Multiplexer and (right) OR gate*

As can be seen from Figures 2.2 and 2.3, representing values as probabilities in stochastic bit-streams lets us implement arithmetic with simple logic gates. For two independent bit-streams X and Y, an AND gate outputs '1' only if both inputs are '1', so its output probability is$P(X \wedge Y) = P(X)\,P(Y)$

which realizes multiplication in the stochastic domain [9].

Likewise, a 2-to-1 multiplexer with select stream R produces output Z according to $P(Z) = P(R \wedge X) + P(\neg R \wedge Y) = P(R)\,P(X) + \big(1 - P(R)\big)\,P(Y)$.

By choosing $P(R) = 0.5$ (Figure 2.3), the mux computes $P(Z) = 0.5\,P(X) + 0.5\,P(Y)$,

i.e. the (unweighted) sum of $P(X)$ and $P(Y)$, thereby performing addition [6].

An alternative way to add two probabilities is with an OR gate, whose output probability is$P(X \vee Y) = P(X) + P(Y) - P(X \wedge Y) = P(X) + P(Y) - P(X)\,P(Y)$,

subtracting the overlap so the result stays in 0, 1 [9]. In practice, however, OR-based addition becomes increasingly inaccurate as more bit-streams are combined (see Figure 2.2).

Beyond multiplication and addition, SC can be extended to perform other mathematical operations, which may be considered for future enhancements of this project[13]:

- Subtraction: An EX-OR gate processes bipolar-encoded streams, flipping bits where inputs differ to produce the difference.
- Division: A D flip-flop with a feedback loop, controlled by the divisor, regulates the stochastic bitstream to approximate the quotient.
- Exponentiation: Cascaded AND gates repeatedly filter a stochastic stream, scaling its probability down to approximate exponentiation.
- Logarithms: A unary counter with a comparator tracks pulse accumulation, estimating logarithms based on when the count reaches a threshold.

After computations are performed in the stochastic domain, the results must be translated back into standard binary form to interface with conventional digital systems. This is achieved using a binary counter, which tallies the number of "1"s (denoted as **N**) in a stochastic bit-stream of length **L**. The ratio of ones to the total stream length gives the estimated probability $P = \frac{N}{L}$, which represents the value encoded by the bit-stream. In unipolar encoding, where valid values lie between 0 and 1, this probability directly corresponds to the numerical result of the computation.

However, since binary hardware works with fixed-width integers such as 8-bit values, this probability must be scaled to match the desired output range. The final binary value B is obtained by multiplying the probability by the maximum value representable in n bits, using the formula $B = P\,(2^n - 1)$.

This scaling factor ensures that the output remains within the appropriate digital range and preserves the relative accuracy of the stochastic computation. For example, converting to 8-bit binary involves a scaling factor of 255, since $2^8 - 1 = 255$.

This conversion process is critical for integrating stochastic accelerators into mixed-signal or binary-dominant systems. By accurately mapping stochastic values back to binary, the system maintains compatibility with traditional processors while still benefiting from the power efficiency, noise resilience, and simplicity of stochastic logic. It allows stochastic computing to function as a viable low-power alternative within hybrid architectures, especially in applications where a slight trade-off in precision is acceptable.

While SC circuits have simple hardware requirements, challenges remain, including correlations between input bit-streams and the need for longer bit-streams to achieve higher precision. For instance, matching the precision of an 8-bit binary system often requires bit-streams of 256 bits [6] or more, limiting SC to low-precision tasks. Recent interest in energy-efficient computing has expanded SC applications beyond neural networks to areas like image processing, edge detection. [6] SC's capability to balance energy efficiency and accuracy is especially valuable for resource-constrained environments such as medical implants.

Developing specialised SC-based MAC accelerators optimises convolutional layers in CNNs (convolutional neural networks), addressing challenges such as random fluctuation errors and high latency[9, 14]. These advancements enhance the feasibility of SC for real-world applications, enabling energy-efficient solutions on edge devices.

2.2 RISC-V Processors and FPGA

The integration of RISC-V architectures with hardware accelerators on FPGA systems is an extremely useful advancement in embedded computing [7]. This approach takes advantage of the simplicity, scalability, and modular design of RISC architectures alongside the parallel processing capabilities of hardware accelerators to deliver exceptional performance and energy efficiency.

2.2.1 RISC-V

The open-source, modular nature of RISC-V builds on a history of innovation in processor design. First developed in 2010 at the University of California, Berkeley, RISC-V was designed as a simple, clean-slate Instruction Set Architecture (ISA) that prioritised extensibility and openness [14]. Unlike proprietary architectures, RISC-V was created to be modular and customisable, enabling developers to adapt it for a wide range of applications, from embedded systems to high-performance computing.

This extensibility has made RISC-V particularly well-suited for integrating domain-specific accelerators. Its open ISA allows developers to define custom extensions without sacrificing compatibility with the standard base instructions [14]. For example, scalable vector processing units, a hallmark of modern RISC-V implementations, enhance data-level parallelism when deployed on FPGAs. These units can adjust to workload requirements, improving computational throughput, resource utilisation, and operating frequency.

An increase in educational materials centred on RISC-V architecture has also resulted from its increasing use. RISC-V and FPGA development training is offered by a number of university programs and industry-led projects, giving professionals and students practical experience in hardware-software co-design. For example, the Imagination University Program provides RVfpga, a course that teaches computer architecture using System-on-Chip (SoC) designs and commercial RISC-V cores, as a result of the growing interest in RISC-V as a platform for both academic research and real-world application, universities and research institutions around the world have adopted similar initiatives[2, 15].

The modular approach of RISC-V draws from decades of experience in processor design, combining lessons from previous RISC architectures with a forward-looking vision of flexibility and interoperability. Today, RISC-V's open and extensible design makes it a powerful platform for co-designing hardware and software, particularly in applications requiring tightly coupled accelerators like stochastic computing units or AI processing modules.

2.2.2 FPGA

Field-Programmable Gate Arrays (FPGA) are reprogrammable hardware platforms that can be configured to implement a wide range of digital circuits. This flexibility makes them highly useful for prototyping and developing digital systems.

RISC-V implementations on an FPGA have been widely explored in previous research and projects. As an example, the Rocket Chip Generator, developed by UC Berkeley [16], uses a configurable RISC-V core that is synthesizable on an FPGA. Used to support features like configurable memory hierarchies, optional floating-point units, and vector processing, this flexibility has made RISC-V a popular choice for prototyping and experimenting with advanced processor features on FPGA platforms.

The modular nature of RISC-V enables custom accelerators to be seamlessly integrated into the processor pipeline or connected as coprocessors, reducing data transfer delays and enhancing computational efficiency for specialised tasks when modified. This promotes rapid testing, debugging, and iteration of designs, making RISC-V an excellent choice for building scalable, efficient, and application-specific solutions on FPGAs.

3 Economic, Legal, Social, Ethical and Environmental Context (ELSEE)

This project's scope introduces a range of considerations beyond technical feasibility, encompassing economic, legal, social, ethical, and environmental aspects. Economically speaking, the use of stochastic computing has the potential to drastically cut chip area and power consumption [11], which would lower manufacturing and operating expenses. Conventional computing architectures frequently call for hardware that uses many resources, increasing the complexity of manufacturing and energy requirements. Stochastic computing, on the other hand, makes use of simplified arithmetic operations to lower circuit complexity and boost efficiency. Furthermore, RISC-V's open-source nature removes licensing costs, promoting accessibility and stimulating creativity among smaller businesses and academic institutions.

The open-standard nature of RISC-V allows for a wide range of customisation, but it's crucial to make sure that intellectual property laws are respected since most reputable organisations have clear policies of not infringing valid competitor IP . Any enhancements or modifications to existing methods must be evaluated to avoid patent infringement. Additionally, FPGA-based implementations must comply with recognised hardware design standards and legal requirements to ensure safety, security, and dependability in real-world applications.

The societal impact of this advancement extends to industries that rely on efficient, low-power computing solutions, such as embedded systems, telecommunications, and real-time signal processing. As computing demands exponentially grow, especially in data-intensive fields, e.g. AI, there is increasing pressure to develop hardware capable of balancing performance with energy efficiency [17]. Stochastic computing is especially promising in this regard, enabling more power-efficient implementations.

Ethical considerations must also be acknowledged, particularly regarding the long-term implications of integrating approximate computing methods into critical applications. Even though stochastic computing is able to increase efficiency, if it is ever to be used in safety-critical applications like secure communications, automotive systems, or aerospace, the inherent trade-off in computational accuracy must be carefully managed to ensure reliability. Applications that leverage stochastic computing must also be designed and tested rigorously to avoid unintended biases by using inaccurate calculations. Validation testing and adherence to industry standards are required to mitigate risks associated with inaccuracies in computation.

Environmental effects remain a key downside of computing hardware development, particularly as the demand for processing power continues to grow [17]. High-performance computing systems are increasingly using a high amount of energy from the power grid. The implementation of energy-efficient hardware designs, such as stochastic computing, offers a potential solution by allowing for hardware that uses reduced power and minimises thermal output, decreasing the overall energy footprint required to power and cool these chips. Additionally, the reconfigurability of FPGA hardware extends the lifespan of computing components, reducing the need for frequent hardware replacements and mitigating electronic waste (e-waste).

4 Methodology

The design process commenced with the implementation of a conventional pipelined MAC unit, serving as a performance benchmark for traditional arithmetic operations. Building on this foundation, a SCAU was developed to perform arithmetic using probabilistic bitstreams, incorporating three primary components: a Binary-to-Stochastic Number Generator, Stochastic Arithmetic Units (comprising multipliers and adders), and a Stochastic-to-Binary Conversion Unit. Following successful verification, the SCAU was integrated into a RISC-V processor as a memory-mapped peripheral, enabling direct communication with the CPU via the Wishbone bus. This required modifications to the processor's hardware, along with the development of software routines to configure and control the accelerator. The subsequent sections detail each stage of this design and integration process.

4.1 Design Methodology

4.1.1 SCAU Design Stage

The design stage is outlined with a simple block diagram of the SCAU (Figure 4.1). The SCAU needs to be able to convert an 8-bit binary operand into a stochastic bit stream of variable length, then carry out a multiply-accumulate calculation with basic logic gates, and then count the resulting "1"s and scale them up to get an approximate binary result. Figure 4.1 shows these three modules, Binary-to-Stochastic, Stochastic Arithmetic, and Stochastic-to-Binary, working sequentially.

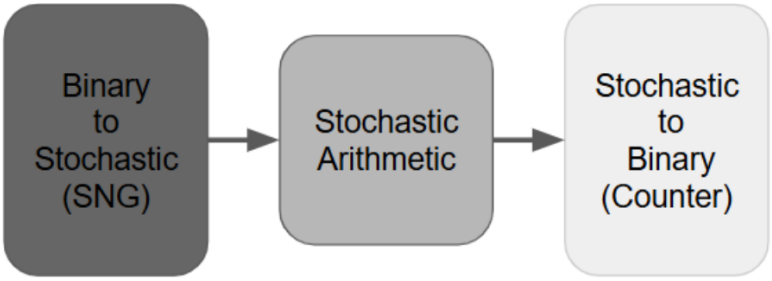

Figure 4.1: *Flow chart showing data flow of SCAU.*

4.1.2 Binary-to-Stochastic Number Generator (SNG)

This design adopts a conventional stochastic-number generator in which an 8-bit maximal-length linear-feedback shift register (LFSR), seeded uniquely for each channel and tapping bits 8, 6, 5 and 4 to implement the basic polynomial $x^8 + x^6 + x^5 + x^4 + 1$. The LFSR generates a pseudo-random number, which is compared against the binary input to produce the stochastic output. To maintain simplicity and ensure consistent performance, a dedicated SNG is assigned to each binary input as seen in Figure 4.2. While this approach increases resource utilisation, it allows for a more straightforward implementation with improved speed and reduced design complexity.

4.1.3 Stochastic Arithmetic Units (Multiplication and Addition)

In SC, multiplication and addition are performed with simple logic gates as described in Chapter 2. In this project, no standalone adder is employed; instead, the stochastic-to-binary conversion unit's counting unit can also effectively handle adding by having multiple inputs. This capability stems from the arithmetic's inherent simplicity, which also reduces overall chip utilisation.

To illustrate the design's flexibility, parallelisation has been made straightforward and configurable through the use of multiple channels. The implementation can generate several multipliers (AND gates) in parallel, each with its own dedicated SNG for both x and y, enabling simultaneous processing of multiple binary inputs. Consequently, the architecture strikes a versatile balance between resource consumption and performance, rendering it scalable for larger, high-throughput applications while preserving implementation simplicity.

4.1.4 Stochastic-to-Binary Conversion Unit

To convert back to binary, the number of '1s in the stochastic bit stream is counted over a defined window length. This counter-based method provides a binary approximation of the computed value, which is then scaled using a predefined scaling factor.

The scaling factor ensures that all stochastic values remain within the valid range of 0 to 1. Without proper scaling, repeated multiplications and additions could push the values outside this range, resulting in streams of all 1s or all 0s, making the output meaningless. By scaling intermediate values appropriately, the stochastic bit streams preserve accuracy, and the final binary output reliably reflects the intended computation.

The accuracy of this final result depends on the length of the stochastic stream—the longer the stream, the more precise the approximation.

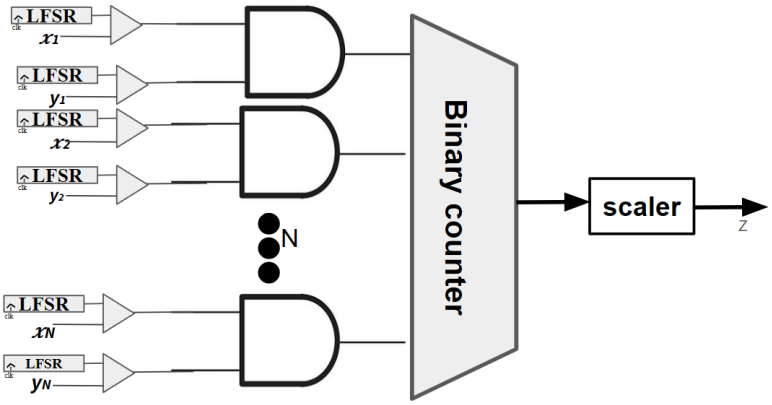

Figure 4.2: *Top-Level Block Diagram of the Stochastic Computing Accelerator Unit (SCAU) .*

4.1.5 Benchmark unit design

A baseline for comparison was required to evaluate the feasibility and performance of the SCAU. A traditional pipelined MAC unit was coded, which acts as a control for benchmarking. This MAC unit is used to assess critical metrics such as speed, power consumption, and chip area. Specifically, the aim is to determine the amount of parallelisation required to achieve similar performance in stochastic computing while evaluating whether the SCAU could offer substantial improvements in power efficiency and area utilisation.

4.2 Integration with RISC-V Processor

During the investigation into integrating the SCAU accelerator with a RISC-V processor, the RVfpga laboratory materials from the Imagination University Program were identified as the most valuable source of guidance and information. The 20-lab series offers step-by-step guidance on customising and extending the SweRVolfSoC RISC-V core, complete with example code, FPGA implementation workflows, and clear instructions for modifying the core whenever necessary [2].

4.2.1 Integration Steps

The process of integrating a custom hardware accelerator into the RVfpga platform is a structured but complex task. Guidance on this procedure is provided in Labs 5–10 of the RVfpga teaching materials, which cover topics including hardware design, bus protocol interfacing, memory mapping, and top-level system integration.

As explored in this project, the integration workflow begins with setting up the RISC-V CPU design in Xilinx Vivado, following Lab 5's guidance. This typically includes initiatimng the SweRVolfSoC core alongside on-chip memory and configuring the Wishbone interconnect, a standard protocol for communication between the processor and peripheral devices. Once synthesis and timing analysis are complete, a bitstream can be generated and programmed onto the FPGA to verify basic CPU operation before proceeding with integration work.

To enable compatibility with the Wishbone bus, a custom accelerator would require a protocol-compliant wrapper module. This wrapper, designed in SystemVerilog, bridges the accelerator's internal signals with the Wishbone interface, handling clock/reset signals, address and data lines, control lines such as write enable, and standard handshake signals like strobe and acknowledge. A copy of the code is in Appendix A.

The integration approach involves modifying the Wishbone interconnect configuration files (wb_intercon.v and .vh) to include the accelerator as a new slave device. This includes assigning a unique memory-mapped address range (Table 4.1) and updating decoding logic using MATCH_ADDR and MATCH_MASK parameters to ensure proper routing of processor transactions.

At the top level, the accelerator would be initiated in the SweRVolfSoC_core.v file, with its Wishbone connections explicitly defined to reflect the expected data flow as seen in Figure 4.3. Control and status registers could then be mapped to accessible memory addresses, allowing the processor to interact with the accelerator via standard read/write instructions. Internally, a case-based decoder can be used to interpret incoming addresses and route them to the appropriate registers based on control logic.

Register	**Base Address**	**Type**	**Bits**	**Description**
ACCEL_DATA	0x80003200	Read/Write	32	Write input data (X or Y) to accelerator
ACCEL_CTRL	0x80003204	Write	32	Start signal to begin computation
ACCEL_RESULT	0x80003208	Read	32	Result of computation from accelerator
ACCEL_STATUS	0x8000320C	Read	8	Stores data from slave Status register

Table 4.1: *Memory map of the SCAU Wishbone interface*

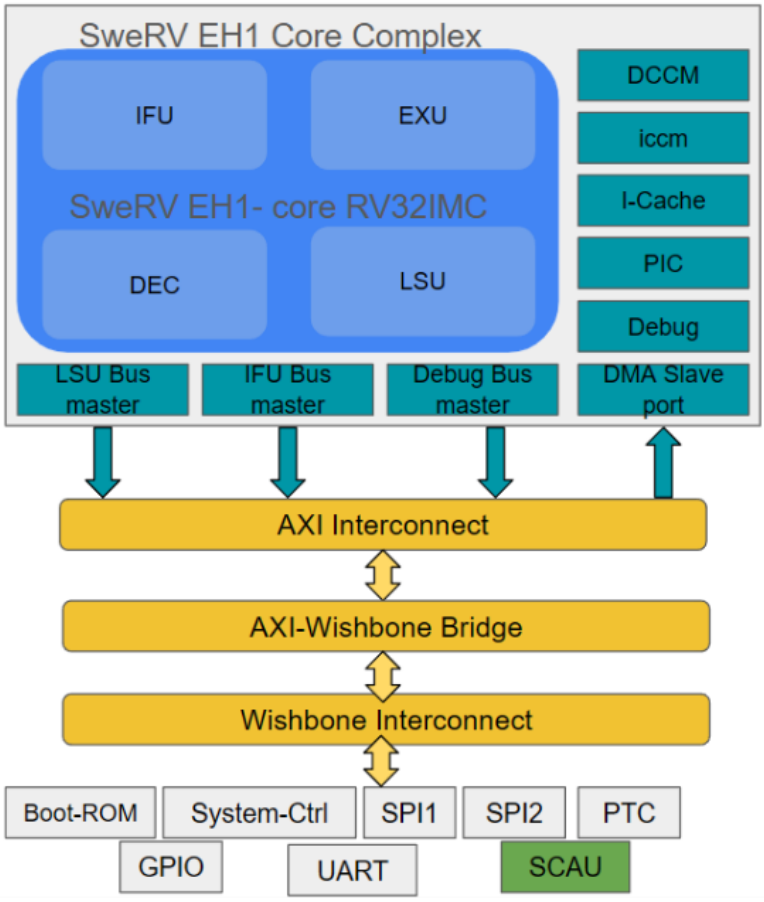

Figure 4.3: *RISC-V Wishbone interface of proposed final desighn*

4.2.2 Software Support

The development environment most commonly used for this work is PlatformIO, following guidance from the RVfpga labs. As outlined in these materials, software control of a custom accelerator integrated into a RISC-V system typically involves interaction via memory-mapped registers, accessed using volatile pointers or macros in C. This method is designed to prevent the compiler from optimising away critical hardware interactions.

A typical control flow (copy of actual code used in Appendix A), which was explored in this investigation, includes:

1. Loading data into the accelerator's registers;
2. Triggering computation by setting specific control bits;
3. Monitoring a status register or waiting for an interrupt to signal completion.

This approach supports strict ordering of operations and helps maintain real-time synchronisation between the processor and the accelerator. Although full software integration was not realised within this project, the investigation outlined the expected structure and logic required for cohesive control of heterogeneous computing elements within a reconfigurable FPGA-based system.

5 Results and Evaluation

All testing presented in this section was carried out using simulation and hardware implementation workflows within the Xilinx Vivado Design Suite. Simulations were conducted under the default conditions provided by Vivado, ensuring consistency in timing analysis, logic behaviour, and resource estimation. This approach enabled both module-level and system-level evaluation of the proposed stochastic computing accelerator design, from functional correctness to integration within the RISC-V processor framework.

5.1 Module-Level Verification

5.1.1 Binary-to-Stochastic Number Generator (SNG)

The protocol for testing for the SNG (Appendix B.1) was to input a binary number and to read the 256 stochastic bitstream output, This is then compared against the expected value, within a ±0.05 tolerance range. This method verifies the SNG's correctness and confirms the LFSR's statistical quality alongside the comparator's functionality. An example output should look like this: "Complete Bit Stream for TEST_INPUT = 255:11111111. . . ..11111, PASS: Bitstream validation passed. Collected ones = 255, Expected ones = 255.00"

These tests were then performed for every possible integer input value between 0-255 to verify that this module works as expected for all possible outputs. Figure 5.1 plots these resulting outputs with the x-axis representing the position of each bit in the bitstream in order and the y-axis showing the sum of number of "1"s counted in that specific bitstream position over all of the 256 bitstreams.

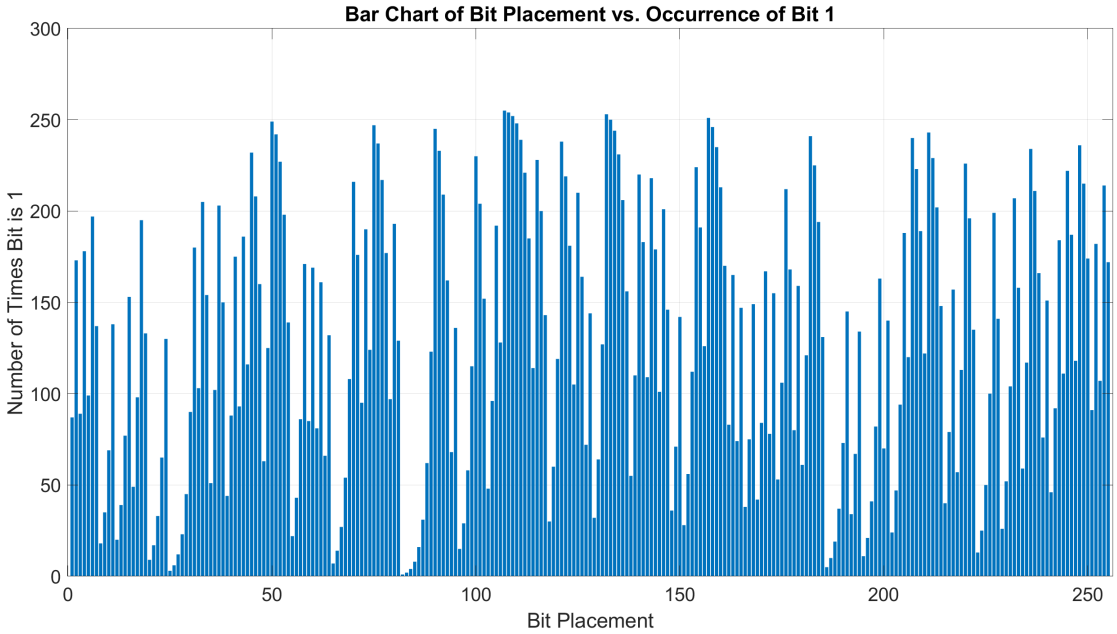

Figure 5.1: *Output from the randomness test of the SNG shown by plotting the summed spread of 0s and 1s for number input 1- 255.*

The results of these tests are shown in Figure 5.1, and it can be seen to demonstrate consistent reliability. Generating accurate stochastic bitstreams within acceptable tolerance ranges. However, as shown in Figure 5.1, pseudo-random number methods, such as the Linear Feedback Shift Register (LFSR), while being a practical and realistic approach for generating random bitstreams. it does exhibit inherent limitations: The supposedly 'random' distribution of 0s and 1s in Figure 5.1 clearly shows periodic peaks and troughs. Since in an ideal random number generator, it should be effectively even distribution across the whole range highlighting the deterministic nature of pseudo-random number generation.

5.1.2 Stochastic Arithmetic Units (Multiplication and Addition) / Stochastic-to-Binary Conversion Unit

The stochastic-arithmetic and conversion stages were merged into a single module, and a unified testbench (see Appendix B) verified both its arithmetic operations and the subsequent binary reconversion. Predefined bit-stream inputs, each with a known expected result were fed into the module, where they underwent multiplication and accumulation before being converted back to binary for comparison. Expected results are compared with each predicted output, and if this is within acceptable ranges, it passes with a typical outcome, which would look for example like "PASSED: result = x, expected = x" This exercise was performed done randomly for multiple different inputs to test the design in order to verify that it works.

5.2 System-Level SCAU Verification and Assessment

Following the successful verification of individual hardware modules, system-level testing was conducted to evaluate the performance of the integrated SCAU. This focused on assessing the overall accuracy of computation and the impact of increasing the pluralisation of the chip on the power and timing, as well as the error percentages across a range of inputs. The evaluation utilised a series of tailored test benches designed to expose the system to realistic and edge-case scenarios.

5.2.1 Accuracy and Error Behaviour

To assess computational accuracy and identify potential error patterns, the SCAU, using a bit stream length of 256, was tested over 10,0000 times with random inputs. For each test, the percentage error difference between the stochastic output and the expected result was calculated and recorded. These errors were then plotted in a bar chart (Figure 5.2), illustrating the distribution and structure of deviations across the full range of input conditions. This visualisation makes it easy to spot any systematic biases introduced by the stochastic approximation techniques.

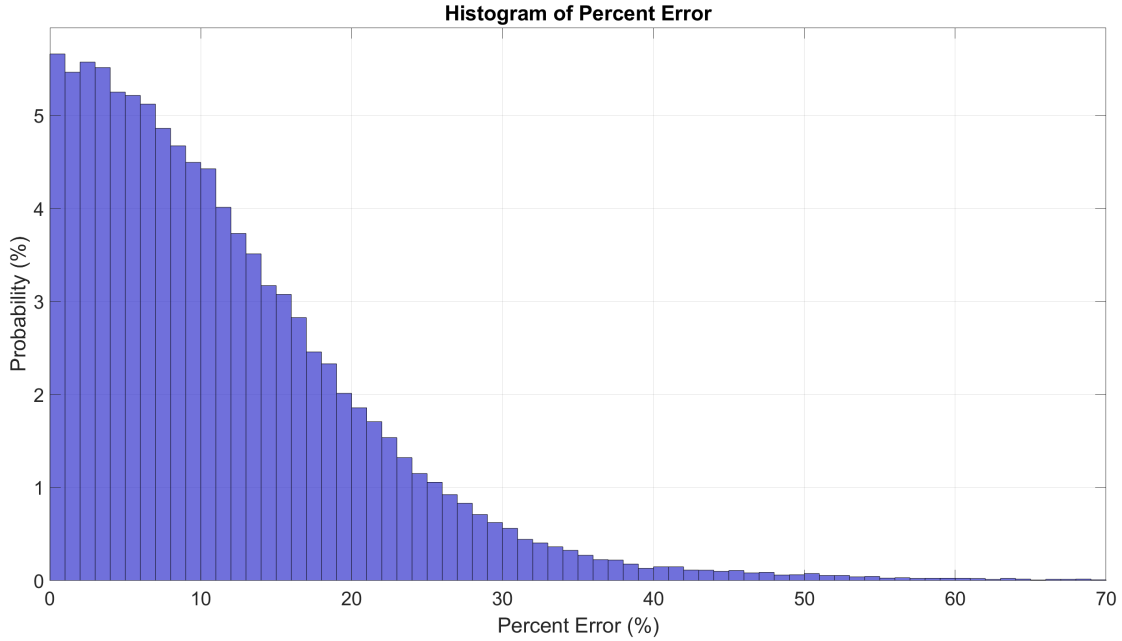

Figure 5.2: *Histogram of Percentage-Error Distribution for SCAU.*

As shown in Figure 5.2, the histogram appears to be an unimodal and positively skewed distribution. The highest concentration of results lies within the lower error range, with per cent errors below 10% occurring most frequently. The frequency of higher errors gradually decreases, forming a long tail that extends toward larger values. This distribution suggests that while the SCAU typically performs with high accuracy, a subset of computations results in larger deviations, which is characteristic of stochastic processing. These higher error percentages are most noticeable when the numbers being multiplied are small, as can be seen in Figure 5.3, which is skewed with the number of outliers being vastly more when the expected result is less.

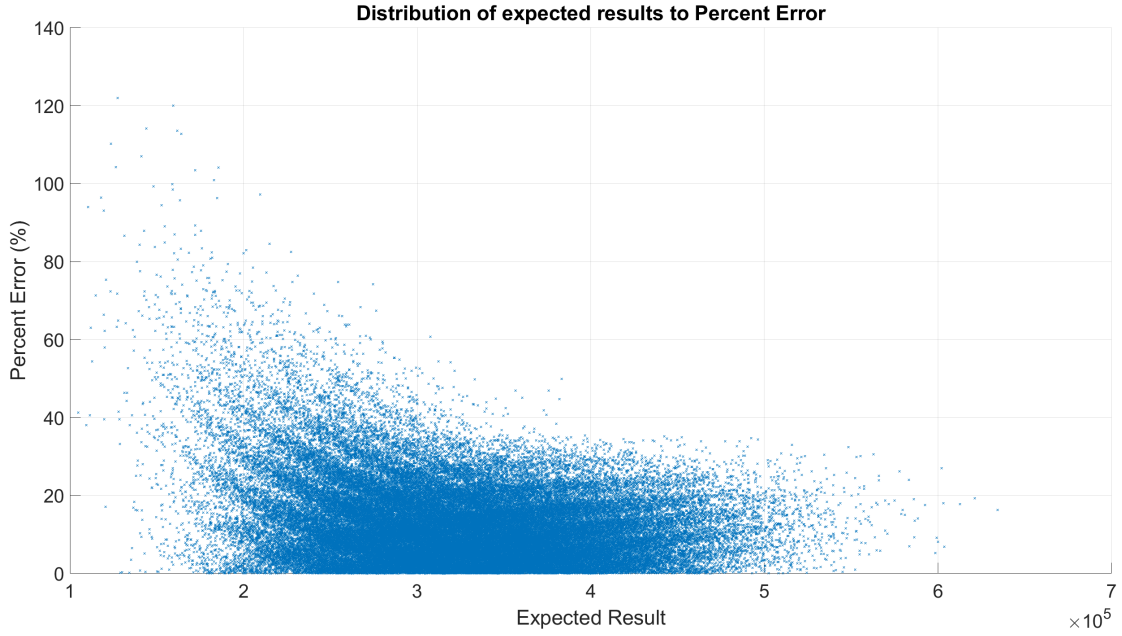

Figure 5.3: *Scatter Plot of Error vs. Expected Result in Stochastic Computations.*

A further factor affecting the accuracy of the output is the length of the bitstream used to run these calculations. As shown in Figure 5.4, the graph only becomes reliably accurate when the bitstream is 256 bits long (2^8), as each binary possibility is represented at that length. However, if you increase the bitstream length further, the graph continues to decline and plateaus around the 2% error mark. This is unexpected, as it should theoretically asymptote towards 0%. This discrepancy may be caused by pseudo-random noise, which could be affecting the accuracy of the results. Further testing would be needed to verify the true cause.

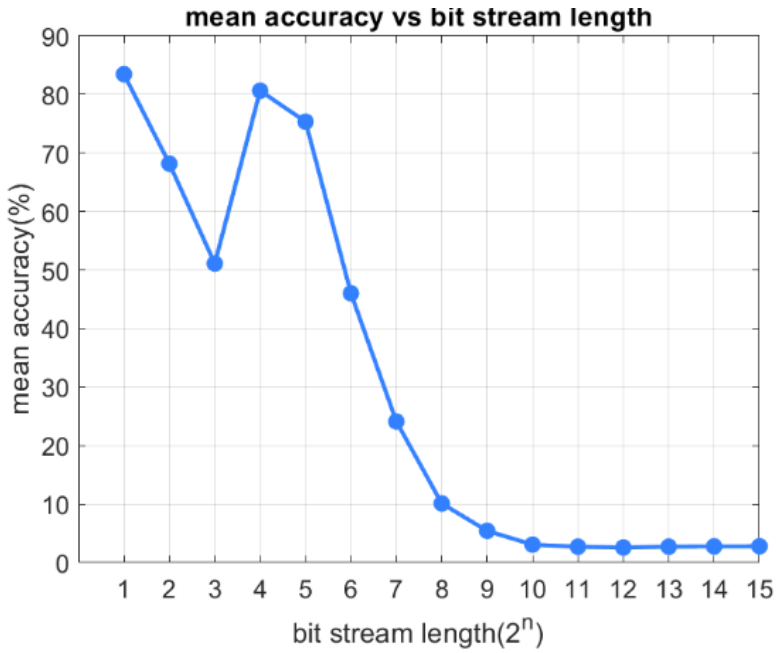

Figure 5.4: *Impact of Bit-Stream Length on Stochastic Computation Error*

5.2.2 Power and Timing Evaluation when Increasing Parallelisation

The energy efficiency of the system was evaluated by graphing the power consumption and execution time per two inputs for typical workloads as parallelisation increased. This analysis quantifies the energy trade-offs involved when scaling up the number of concurrent stochastic channels used in processing. Notably, the SNG, which includes LFSR and comparators, contribute significantly to the overall power consumption. As the number of parallel units grows, so too does the number of SNG required, each consuming additional logic and switching power. These components, while necessary for generating high-quality bitstreams, add a measurable overhead to both power and timing.

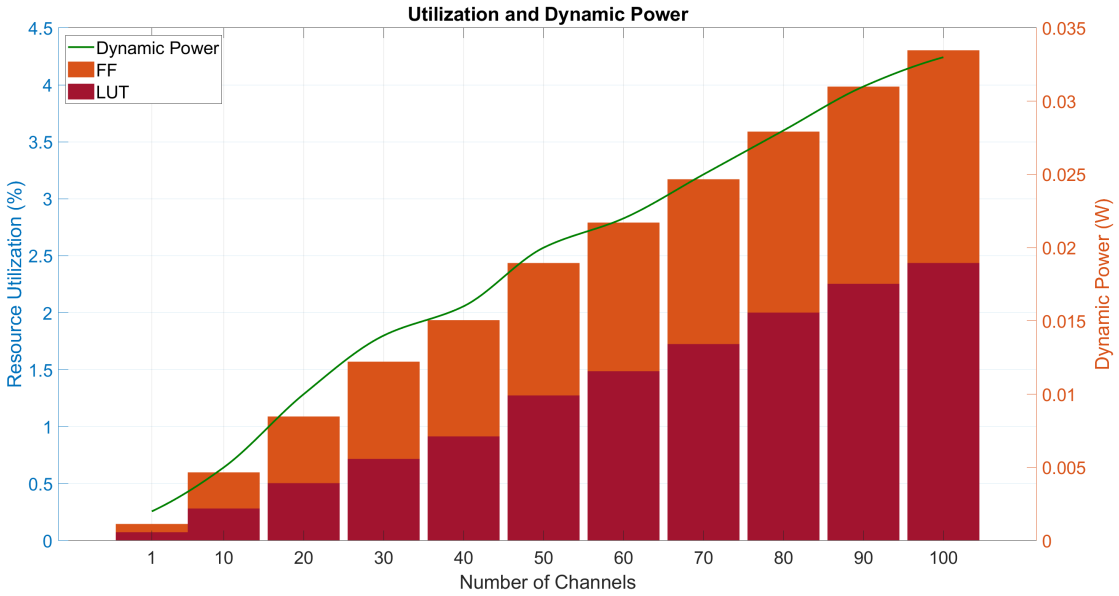

Figure 5.5: *Dynamic Power Consumption vs. Number of Parallel SCAU Channel*

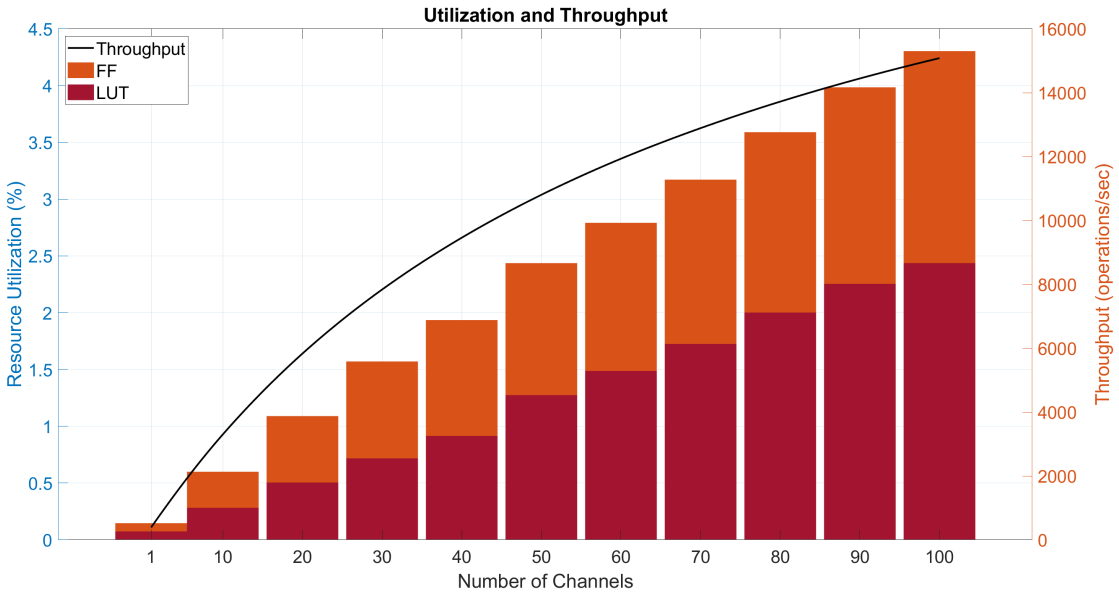

Figure 5.6: *Throughput and Latency Trade-Off with Increasing Parallelisation*

In Figure 5.5, the observed power consumption exhibits an almost linear increase with each additional stochastic channel. This pattern shows that, as one might expect that each newly introduced channel independently contributes to the overall switching activity and, in turn, raises the dynamic power draw. Interestingly, for moderate channel counts, the total power consumption still remains within operational limits of the Nexys A7 development board, indicating that a modest to large amount of parallelism is both feasible and safe, with timing errors only occurring at the 100+ range. Turning to Figure 5.6, the timing and throughput data underscore the trade-off between per-channel latency and aggregate processing capacity. Although each channel must complete a full stochastic bitstream cycle, yielding a fixed per-channel latency, the total throughput increases when multiple channels operate in parallel, effectively handling multiple computations within the same time window. From the collected data, it appears that timing closure is well maintained, indicating that routing congestion and clock-domain contention have not yet become critical bottlenecks. However, should the system integrate a far greater number

of channels, it is plausible that the maximum achievable clock frequency would decline due to more complex routing paths, thus partially offsetting the raw throughput gains, as can be seen by the graph beginning to level out as the number of channels exceeds 100.

Each added channel in the SNG design increases FPGA resource usage, raising dynamic power consumption, on-chip temperatures, and the demand for LUTs, flip-flops, and routing. While moderate channel counts remain within safe limits, further scaling may trigger thermal and routing challenges that could cap clock speeds. Additionally, higher parallelism heightens the risk of correlated outputs if the LFSR isn't properly seeded. Robust seed diversification is therefore crucial to maintaining accuracy as the system scales.

5.3 Benchmark Comparison Testing

To compare the SCAU with a standard deterministic design, a conventional MAC unit was implemented and tested under the same operating conditions. The analysis centred on four main metrics: throughput, power usage, accuracy and precision, and hardware resource utilisation. Both the stochastic and deterministic MAC designs were synthesised and tested on the Nexys A7 FPGA, using various input patterns to evaluate power efficiency and resource requirements. the SCAU has a parallelisation of 5 as the power is too low to be accurately read at 1 for a single input . Results are shown in Table 5.1 below :

Measurement	**Benchmark MACed**	**SCAU**
Dynamic Power (W)	0.094	0.003
Static Power (W)	0.097	0.097
Total On-Chip Power (W)	0.191	0.1
LUTs Used	66	79
LUTRAM Used	33	0
Flip-Flops (FF)	83	214
IOs Used	100	47
BUFGs Used	1	1
Throughput (Inputs/ns)	0.0974	0.0021
Latency (ns)	10.27	466
Energy per Operation (pJ)	966	1398
Average Error Rate (%)	0	1.58

Table 5.1: Comparison between the SCAU compared to Traditional MACed

The benchmark comparison between the SCAU and the benchmark MAC unit highlights a clear trade-off between accuracy, performance, and power efficiency. The deterministic MAC provides perfect precision with a 0% error rate, while the SCAU incurs an average error rate of 1.58%. This slight reduction in accuracy is typical of stochastic computing, which introduces variability due to its probabilistic nature. Despite this, in applications like neural networks, image processing, and probabilistic inference, where approximate results are acceptable, the error is within a tolerable range. From a power perspective, the SCAU is significantly more efficient. It consumes only 0.003 W of dynamic power compared to the 0.094 W of the deterministic MAC, resulting in a much lower total on-chip power (0.1 W vs. 0.191 W). However, due to its higher latency (466 ns vs. 10.27 ns) and lower throughput (0.0021 inputs/ns vs. 0.0974 inputs/ns), the energy per operation is actually higher for the SCAU (1398 pJ vs. 966 pJ). This highlights a key trade-off: while the stochastic logic itself is power-efficient, its slower operation can offset the energy savings in high-performance scenarios. In terms of resource utilisation, the

SCAU avoids using LUTRAM altogether and consumes more flip-flops slightly (214 vs. 83), suggesting it trades memory for additional control logic. It also uses slightly more LUTs (79 vs. 66), indicating a modest increase in logic complexity. A major bottleneck of the design appears to be the SNG unit, as it appears to take the majority of what can be seen from Table 5.1. The increase in chip space usage is mainly due to an increased use of SNG chips, the more channels used to run the calculations. The SCAU demonstrates strong modularity and low dynamic power consumption, making it attractive for low-power, error-tolerant applications. However, its increased latency and lower throughput may limit its applicability in high-speed pipelines unless parallelism is exploited. While not a universal replacement for deterministic MACs, the SCAU is a promising domain-specific accelerator that complements traditional designs in scenarios where energy efficiency and acceptable approximation are priorities.

5.4 Testing and Validation of the RISC-V Integrated Accelerator

5.4.1 Testing of the Wishbone interface

A dedicated SystemVerilog testbench was created to verify the Wishbone integrated accelerator by systematically feeding input values that the RISC-V core would input and capturing the corresponding outputs. This approach checks a range of input scenarios from boundary cases to typical operational values and automatically compares the accelerator's results against the expected outcomes. This approach decreases the possibility of bugs occurring after integration, thereby streamlining the process of identifying bugs or mismatches. A copy of the example waveform is in Appendix C, Figure 1. Although this plan was only partially realised in the work described here, the methodology has been shown to be sufficiently robust that it can provide the requested function checks.

5.4.2 Proposed Testing and Validation Plan for RISC-V integrated SCAU

Once integrated as in Appendix C, Figure 2, the evaluation strategy would commence for future work by integrating the SCAU into a RISC-V system for neural network inference tasks. The accelerator would be embedded into a deep learning framework and tested using established image classification datasets such as CIFAR-10 and ImageNet. In this context, key performance metrics including classification accuracy, throughput (measured in images per second), and energy efficiency would be carefully monitored to verify that the stochastic computations are capable of delivering the expected approximation quality and processing speed for modern AI workloads.

Additional evaluations would then be conducted under controlled conditions to simulate real-time applications. For example, the SCAU's performance in processing live data streams representative of simplified signal processing tasks would be examined. In these experiments, performance parameters such as latency, throughput, and frame rate would be measured and compared to those obtained from conventional accelerator configurations. Incrementally deploying multiple stochastic channels would further help assess the effects on power consumption, resource utilisation, and inter-channel correlations insights that would serve as a foundation for design optimisations in the future.

The insights gained from these evaluations will guide design refinements identifying the most suitable application areas and suggesting task specific improvements thereby enabling implementation of this technology for more energy-efficient computations.

6 Conclusions

6.1 Summary of Main Findings

The project set out to explore whether a SCAU could reduce power consumption in data-intensive tasks by accepting a degree of approximation. As a baseline for comparison, a standard MAC unit was implemented to measure performance, accuracy, and resource usage.

The SCAU implemented here demonstrates the conceptual viability of approximate computing in tandem with a standard processor. Utilising simple logic gates for multiplication and accumulation significantly reduces the complexity inherent in full-precision arithmetic units. Moreover, its modular architecture supports the possibility of scaling to multiple parallel channels for higher throughput, albeit at the cost of increased hardware resources.

Results demonstrated that the SCAU consistently consumed less power than the deterministic MAC, albeit with higher latency and moderate error levels. Parallelising the design showed potential for throughput gains, but it also increased resource usage, underscoring a balance between performance scaling and hardware constraints.

Although the accelerator was not fully integrated within the RISC-V core, preliminary work on a Wishbone-compatible interface indicated that integration into an open-source RISC-V system is both practical and flexible. The investigation showed that approximate computing can indeed deliver notable power savings at the expense a slight sacrifice in precision. This suggests that SCAUs may indeed offer a promising approach for energy-sensitive applications, especially those that inherently tolerate a degree of inexactness.

6.2 Limitations and Potential Inaccuracies

While the stochastic computing accelerator generally produced reliable and repeatable outputs, several factors may have contributed to the inaccuracies observed during testing. One key limitation lies in the use of LFSR for pseudo-random number generation. Although efficient and commonly used in hardware implementations, LFSRs are deterministic and can produce repeating or correlated patterns as seen in Figure 5.1, particularly over shorter bitstreams. These correlations can introduce bias, especially in computations involving small input values where the impact of such statistical variation is magnified. In designs where multiple LFSRs are used concurrently, the possibility of pattern alignment or correlation across channels further increases the likelihood of compounded error.

The synthesis and implementation processes within the Xilinx Vivado toolchain may have subtly influenced the accuracy of results. Vivado applies automated optimisations such as logic simplification, register retiming, and timing-driven placement that, while preserving functional correctness, can alter the internal timing and structure of modules [18]. These changes are generally benign in deterministic systems but may disrupt the statistical properties that stochastic computing relies upon, especially in designs sensitive to cycle-level synchronisation of bitstreams. Additionally, post-synthesis simulation tools may not fully capture hardware-level timing mismatches or bitstream accumulation effects, particularly if testbench configurations assume ideal conditions [19]. This becomes especially problematic in low-magnitude results, where small errors introduced through rounding, bit truncation, or limited resolution in measurement tools can disproportionately affect perceived accuracy.

A final consideration relates to the accuracy of the resource utilisation and power estimates provided by Vivado. While these estimates are typically based on default assumptions, in this project, real switching activity data was used to improve the accuracy of the power analysis. Nonetheless, even with realistic activity files, the resulting figures can still be influenced by other factors such as estimated junction temperature, voltage scaling, or clock gating behaviour that may not be fully captured in simulation [20]. These discrepancies can lead to a gap between reported and actual power consumption. Additionally, as this work was completed as part of an undergraduate project, some aspects of the design flow and verification may not fully reflect optimised or industry-standard methods. Despite this, the project provides a strong proof of concept, and the insights gained offer a solid foundation for future development and refinement of stochastic computing hardware.

6.3 Feasibility Assessment and Practical Application

The ability of SC to reduce power consumption and chip area by replacing full-precision arithmetic units with simple logic gates is now well established. At this stage, the question is whether the resulting errors consistently remain within acceptable bounds for real-world use cases. Aiming to evaluate the practical feasibility of deploying the SCAU in scenarios where controlled approximation is tolerable and efficiency gains outweigh the loss in precision.

For edge-AI and IoT inference, those limits are well mapped. Commercial INT8 flows on Google's Edge-TPU, for example, show that stepping down from floating-point to eight-bit arithmetic costs under 1 % top-1 accuracy while more than doubling energy efficiency [21]. Vendors, therefore, treat a $\approx$ 2% % mean-absolute-error budget as effectively lossless; any SC core that stays within that envelope can slot straight into smart speakers, wearables, or camera nodes without retraining.

Real-time imaging is even more forgiving. Hardware trials of edge enhancement and CNN feature extractors find that frames remain visually indistinguishable from the reference when arithmetic error sits between 3 % and 5 % [22]. Staying in that single-digit band lets SC blocks live in the image-signal-processing pipeline, powering AR overlays and low-latency video encoders without visible artefacts.

Finally, ultra-low-power biomedical and environmental sensors could be considered a sweet spot: they aim for $\approx$ 98% classification accuracy ($\approx$ 2% error) yet run off microwatts [23]. Dropping deterministic MACs for SC logic translates straight into longer battery life.

Maintaining stochastic mean error below approximately 2% is generally sufficient for use in edge-AI applications, while tolerances up to around 5% remain acceptable for imaging tasks and probabilistic analysis. Within these bounds, stochastic computing presents a low-overhead, energy-efficient solution for reducing silicon area without significantly impacting functional correctness.

6.4 Future Work

In future work, it would be interesting to implement a dedicated external SNG. By isolating the SNG from the main chip architecture, more advanced and less resource-intensive approaches, such as true random number generators, can be effectively utilised. This, in theory, would help improve bitstream quality, enhancing both the speed and reliability of stochastic computations that have been discussed in Chapter 5.1, as this was definitely a limiting factor in the design performance.

Additionally, further research into optimised parallelisation would provide insight into how far performance gains can be scaled effectively while balancing hardware resource usage, energy efficiency, and computational accuracy. As this project only considered 100 as a maximum, it would be interesting to see what would happen up 256 and potentially get one calculation per clock cycle. This could guide future designs toward achieving higher throughput and power consumption.

Finally, using stochastic methods for different accelerator calculations in Chapter 5.1, such as subtracting, dividing, logarithms, and powers, would significantly enhance its applicability. Incorporating more of these operation types and potentially more would expand the scope of potential applications, as this project only touched upon the very basics, where traditional MAC has an advantage.

REFERENCES

[1] "410-292," DigiKey Electronics. [Online]. Available: https://www.digikey.co.uk/en/products/detail/digilent-inc/410-292/5117190?s=N4IgTCBcDaIGYAcDmBDABAOwKYA8CeAzmigOwgC6AvkA. [Accessed: Apr. 22, 2025].

[2] "Training and Teaching," Imagination University Programme. [Online]. Available: https://university.imgtec.com/teaching-download/. [Accessed: Apr. 17, 2025].

[3] "Downloads," AMD. [Online]. Available: https://www.xilinx.com/support/download.html. [Accessed: Apr. 22, 2025].

[4] PlatformIO, "PlatformIO: Your Gateway to Embedded Software Development Excellence." [Online]. Available: https://platformio.org/. [Accessed: Apr. 22, 2025].

[5] "Global Energy Perspective 2024." [Online]. Available: https://www.mckinsey.com/industries/energy-and-materials/our-insights/global-energy-perspective. [Accessed: Apr. 17, 2025].

[6] A. Alaghi, W. Qian, and J. P. Hayes, "The promise and challenge of stochastic computing," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 37, no. 8, pp. 1515–1531, Aug. 2018. doi:10.1109/tcad.2017.2778107.

[7] M. A. Islam and K. Kise, "Resource-efficient RISC-V vector extension architecture for FPGA-based accelerators," in *Proc. 13th Int. Symp. Highly Efficient Accelerators and Reconfigurable Technologies*, ACM, Jun. 2023, pp. 78–85. doi:10.1145/3597031.3597047.

[8] H. Sim and J. Lee, "A new stochastic computing multiplier with application to deep convolutional neural networks," in *Proc. 54th Annu. Design Autom. Conf.*, ACM, Jun. 2017. doi:10.1145/3061639.3062290.

[9] P. Schober, M. H. Najafi, and N. Taherinejad, "High-accuracy multiply-accumulate (MAC) technique for unary stochastic computing," *IEEE Trans. Comput.*, vol. 71, no. 6, pp. 1–1, Jun. 2021. doi:10.1109/tc.2021.3087027.

[10] V. M. H. Cruz, "Exploring Stochastic Computing Applied to Artificial Perception: A Convolutional Neural Network Implementation on Reconfigurable Logic," Universidade de Coimbra, 2022. [Online]. Available: https://search.proquest.com/openview/4effd40a160361823728fad0b295489e/1?pq-origsite=gscholar&cbl=2026366&diss=y.

[11] B. R. Gaines, "Origins of stochastic computing," in *Stochastic Computing: Techniques and Applications*, Springer, 2019, pp. 13–37. doi:10.1007/978-3-030-03730-7_2.

[12] H. Hsiao, J. Anderson, and Y. Hara-Azumi, "Generating stochastic bitstreams," in *Stochastic Computing: Techniques and Applications*, Springer, 2019, pp. 137–152. doi:10.1007/978-3-030-03730-7_7.

[13] A. Alaghi and J. P. Hayes, "On the functions realized by stochastic computing circuits," in *Proc. 25th Great Lakes Symp. VLSI*, ACM, May 2015. doi:10.1145/2742060.2743758.

[14] Wevolver, "RISC-V architecture: A comprehensive guide to the open-source ISA," 2024. [Online]. Available: https://www.wevolver.com/article/risc-v-architecture. [Accessed: Nov. 25, 2024].

[15] "The RVfpga Teaching Package." [Online]. Available: https://www.techrxiv.org/users/845563/articles/1234034/master/file/data/TheRVfpgaTeachingPackage/TheRVfpgaTeachingPackage.pdf. [Accessed: Apr. 17, 2025].

[16] "Tech reports." [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html. [Accessed: Apr. 16, 2025].

[17] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *IEEE Micro*, vol. 32, no. 3, pp. 122–134, May 2012. doi:10.1109/mm.2012.17.

[18] *Vivado Design Suite User Guide: Implementation (UG904)*, 2024.

[19] P. Limmer, D. Moeller, M. Mueller, and C. Roettgermann, "Validation of Timing Constraints on RTL-Reducing Risk and Effort on Gate-Level," *DVCon Europe*, pp. 1–8, 2016.

[20] *Vivado Design Suite User Guide: Power Analysis and Optimization (UG907)*, v2024.2, "Achieving an Accurate Power Analysis Using Vivado report_power."

[21] M. Baghbanbashi, M. Raji, and B. Ghavami, "Quantizing YOLOv7: A Comprehensive Study," *arXiv [cs.CV]*, Jul. 05, 2024. [Online]. Available: http://arxiv.org/abs/2407.04943.

[22] P. Li and D. J. Lilja, "Using stochastic computing to implement digital image processing algorithms," in *2011 IEEE 29th Int. Conf. Comput. Des. (ICCD)*, IEEE, Oct. 2011. doi:10.1109/iccd.2011.6081391.

[23] F. Sabry, T. Eltaras, W. Labda, K. Alzoubi, and Q. Malluhi, "Machine learning for healthcare wearable devices: The big picture," *J. Healthc. Eng.*, vol. 2022, p. 4653923, Apr. 2022. doi:10.1155/2022/4653923.

Appendix A: Source Code

Appendix A.1: Binary to Stochastic Module

```
`timescale 1ns/1ps
//============================================================
// Module: binary_to_stochastic
// Converts an 8-bit binary value into a stochastic bitstream using
// an 8-bit LFSR (polynomial: x^8+x^6+x^5+x^4+1).
//============================================================
module binary_to_stochastic #(
    parameter BIN_WIDTH = 8,
    parameter ST_LENGTH = 256,
    parameter SEED      = 8'hA5
)(
    input  wire                 clk,
    input  wire                 reset,
    input  wire                 clk_en,
    input  wire [BIN_WIDTH-1:0] binary_input,
    output reg                  stochastic_output
);
    reg [BIN_WIDTH-1:0] lfsr;
    wire feedback = lfsr[BIN_WIDTH-1] ^ lfsr[5] ^ lfsr[4] ^ lfsr[3];

    always @(posedge clk) begin
        if (reset)
            lfsr <= SEED;
        else if (clk_en)
            lfsr <= {lfsr[BIN_WIDTH-2:0], feedback};
    end

    always @(posedge clk) begin
        if (reset)
            stochastic_output <= 1'b0;
        else if (clk_en)
            stochastic_output <= (lfsr < binary_input);
    end
endmodule
```

Listing 1: Binary to Stochastic Conversion Module

Appendix A.2: Stochastic Multiplier, Adder, and Scaler

```
`timescale 1ns/1ps
//============================================================
// Module: sc_multiplier_adder_counter
// Performs the stochastic multiply-accumulate operation.
//============================================================
module sc_multiplier_adder_counter #(
    parameter BIN_WIDTH    = 8,
    parameter ST_LENGTH    = 256,
    parameter NUM_CHANNELS = 100
)(
    input  wire         clk,
    input  wire         reset,
    input  wire         clk_en,
    input  wire         start_run,
    input  wire [NUM_CHANNELS-1:0] stochastic_x,
    input  wire [NUM_CHANNELS-1:0] stochastic_y,
    output reg  [31:0]  result,
```

```
    output reg          valid
);
    localparam SCALE = (1 << (2 * BIN_WIDTH));

    reg [31:0] accumulator;
    reg [$clog2(ST_LENGTH)-1:0] cycle_count;
    reg running;
    integer ch;
    reg [31:0] cycle_sum;

    always @* begin
        cycle_sum = 0;
        for (ch = 0; ch < NUM_CHANNELS; ch = ch + 1) begin
            cycle_sum = cycle_sum + (stochastic_x[ch] & stochastic_y[ch]);
        end
    end

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            accumulator <= 0;
            cycle_count <= 0;
            result      <= 0;
            valid       <= 0;
            running     <= 0;
        end else begin
            valid <= 0;
            if (start_run && !running) begin
                running     <= 1;
                accumulator <= 0;
                cycle_count <= 0;
            end else if (running && clk_en) begin
                if (cycle_count == ST_LENGTH - 1) begin
                    accumulator <= accumulator + cycle_sum;
                    result  <= ((accumulator + cycle_sum) * SCALE) /
                        ST_LENGTH;
                    valid   <= 1;
                    running <= 0;
                end else begin
                    accumulator <= accumulator + cycle_sum;
                    cycle_count <= cycle_count + 1;
                end
            end
        end
    end
endmodule
```

Listing 2: Stochastic Multiplier-Adder-Counter Module

A.3 Stochastic Multiplier-Adder Top Module

```

//=============================================================
// Module: stochastic_multiplier_adder_top_serial
// Top-level wrapper for multiple stochastic MAC channels.
//=============================================================
module stochastic_multiplier_adder_top_serial #(
    parameter BIN_WIDTH    = 8,
    parameter ST_LENGTH    = 256,
    parameter NUM_CHANNELS = 100
)(
    input  wire [31:0] in_data_bus,
```

```
    input  wire [$clog2(NUM_CHANNELS)-1:0] channel_select,
    input  wire        write_en,
    input  wire        write_xy_sel,
    input  wire        start_op,
    output wire [31:0] binary_result,
    output wire        valid,
    input  wire        clk,
    input  wire        reset
);
    reg [BIN_WIDTH-1:0] x_regs [0:NUM_CHANNELS-1];
    reg [BIN_WIDTH-1:0] y_regs [0:NUM_CHANNELS-1];
    integer ch;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            for (ch = 0; ch < NUM_CHANNELS; ch = ch + 1) begin
                x_regs[ch] <= 0;
                y_regs[ch] <= 0;
            end
        end else if (write_en) begin
            if (!write_xy_sel)
                x_regs[channel_select] <= in_data_bus[BIN_WIDTH-1:0];
            else
                y_regs[channel_select] <= in_data_bus[BIN_WIDTH-1:0];
        end
    end

    reg start_run;
    reg power_gated;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            start_run   <= 0;
            power_gated <= 0;
        end else if (start_op) begin
            start_run   <= 1;
            power_gated <= 1;
        end else if (valid) begin
            start_run   <= 0;
            power_gated <= 0;
        end
    end

    wire [NUM_CHANNELS-1:0] stoch_x;
    wire [NUM_CHANNELS-1:0] stoch_y;

    genvar i;
    generate
        for (i = 0; i < NUM_CHANNELS; i = i + 1) begin : GEN_STOCH
            binary_to_stochastic #(
                .BIN_WIDTH(BIN_WIDTH),
                .ST_LENGTH(ST_LENGTH),
                .SEED(8'hA5 + i)
            ) b2s_x (
                .clk(clk),
                .reset(reset),
                .clk_en(power_gated),
                .binary_input(x_regs[i]),
                .stochastic_output(stoch_x[i])
            );

            binary_to_stochastic #(
                .BIN_WIDTH(BIN_WIDTH),
```

```
                .ST_LENGTH(ST_LENGTH),
                .SEED(8'h5A + i)
            ) b2s_y (
                .clk(clk),
                .reset(reset),
                .clk_en(power_gated),
                .binary_input(y_regs[i]),
                .stochastic_output(stoch_y[i])
            );
        end
    endgenerate

    sc_multiplier_adder_counter #(
        .BIN_WIDTH(BIN_WIDTH),
        .ST_LENGTH(ST_LENGTH),
        .NUM_CHANNELS(NUM_CHANNELS)
    ) mac_counter (
        .clk(clk),
        .reset(reset),
        .clk_en(1'b1),
        .start_run(start_run),
        .stochastic_x(stoch_x),
        .stochastic_y(stoch_y),
        .result(binary_result),
        .valid(valid)
    );
endmodule
```

Listing 3: Top level module combining the binary to stochastic the asthmatic unit and stocastic to binary units

A.4 Wishbone Wrapper for Stochastic Accelerator

```
%module accel_wb_wrapper #(
    parameter BIN_WIDTH    = 8,
    parameter ST_LENGTH    = 256,
    parameter NUM_CHANNELS = 1
)(
    input  wire         clk,
    input  wire         reset,
    // Wishbone slave interface signals
    input  wire [31:0]  wb_adr_i,   // Byte address
    input  wire [31:0]  wb_dat_i,
    output reg  [31:0]  wb_dat_o,
    input  wire         wb_we_i,
    input  wire         wb_stb_i,
    input  wire         wb_cyc_i,
    output reg          wb_ack_o,

    // Accelerator interface signals
    output reg [31:0]   accel_in_data_bus,
    output reg [4:0]    accel_channel_select,
    output reg          accel_write_en,
    output reg          accel_write_xy_sel,
    output reg          accel_start_op,
    input  wire [31:0]  accel_binary_result,
    input  wire         accel_valid
);

    // Address offsets
    localparam ADDR_DATA   = 4'h0;  // 0x0: Data register (write-only)
```

```
    localparam ADDR_CTRL   = 4'h4;  // 0x4: Control register (write-only)
    localparam ADDR_RESULT = 4'h8;  // 0x8: Result register (read-only)
    localparam ADDR_STATUS = 4'hC;  // 0xC: Status register (read-only)

    reg done_latched;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            wb_ack_o             <= 1'b0;
            wb_dat_o             <= 32'b0;
            accel_in_data_bus    <= 32'b0;
            accel_channel_select <= 5'b0;
            accel_write_en       <= 1'b0;
            accel_write_xy_sel   <= 1'b0;
            accel_start_op       <= 1'b0;
            done_latched         <= 1'b0;
        end else begin
            // Default: no new write pulses
            accel_write_en <= 1'b0;
            accel_start_op <= 1'b0;

            // Latch the single-cycle valid
            if (accel_valid)
                done_latched <= 1'b1;

            if (wb_stb_i && wb_cyc_i && !wb_ack_o) begin
                case (wb_adr_i[3:0])
                    ADDR_DATA: begin
                        if (wb_we_i) begin
                            // bits:
                            //   [7:0] => data
                            //   [12:8] => channel
                            //   [13] => write_xy_sel
                            accel_in_data_bus    <= {24'b0, wb_dat_i[7:0]};
                            accel_channel_select <= wb_dat_i[12:8];
                            accel_write_xy_sel   <= wb_dat_i[13];
                            accel_write_en       <= 1'b1; // 1-cycle pulse
                        end
                    end

                    ADDR_CTRL: begin
                        if (wb_we_i) begin
                            // bit0 => start
                            if (wb_dat_i[0]) begin
                                accel_start_op <= 1'b1;
                                done_latched   <= 1'b0; // clear old done
                            end
                            // bit1 => manual clear of done
                            if (wb_dat_i[1]) begin
                                done_latched <= 1'b0;
                            end
                        end
                    end

                    ADDR_RESULT: begin
                        // read final result
                        wb_dat_o <= accel_binary_result;
                    end

                    ADDR_STATUS: begin
                        // bit0 => done_latched
                        wb_dat_o <= {31'b0, done_latched};
                    end
```

```
92
93                      default: begin
94                          wb_dat_o <= 32'b0;
95                      end
96                  endcase
97
98                  wb_ack_o <= 1'b1;
99              end else begin
100                 wb_ack_o <= 1'b0;
101             end
102         end
103     end
104
105 endmodule
```

Listing 4: Wishbone-Compliant Wrapper Module

A.5 Top-Level Integration with Wishbone Interface

```
1  `timescale 1ns/1ps
2  module accel_integration_top #(
3      parameter BIN_WIDTH    = 8,
4      parameter ST_LENGTH    = 256,
5      parameter NUM_CHANNELS = 1
6  )(
7      // Wishbone slave interface signals (connect to SoC interconnect)
8      input  wire         clk,
9      input  wire         reset,
10     input  wire [31:0]  wb_adr_i,
11     input  wire [31:0]  wb_dat_i,
12     output wire [31:0]  wb_dat_o,
13     input  wire         wb_we_i,
14     input  wire         wb_stb_i,
15     input  wire         wb_cyc_i,
16     output wire         wb_ack_o,
17
18     // Debug output (e.g. connect to onboard LEDs)
19     output wire [7:0]   debug_leds
20 );
21
22     // Internal signals between the wrapper and the accelerator core
23     wire [31:0] accel_in_data_bus;
24     wire [4:0]  accel_channel_select;
25     wire        accel_write_en;
26     wire        accel_write_xy_sel;
27     wire        accel_start_op;
28     wire [31:0] accel_binary_result;
29     wire        accel_valid;
30
31     // Instantiate the Wishbone wrapper
32     accel_wb_wrapper #(
33         .BIN_WIDTH(BIN_WIDTH),
34         .ST_LENGTH(ST_LENGTH),
35         .NUM_CHANNELS(NUM_CHANNELS)
36     ) U_accel_wb (
37         .clk(clk),
38         .reset(reset),
39         .wb_adr_i(wb_adr_i),
40         .wb_dat_i(wb_dat_i),
41         .wb_dat_o(wb_dat_o),
42         .wb_we_i(wb_we_i),
```

```
        .wb_stb_i(wb_stb_i),
        .wb_cyc_i(wb_cyc_i),
        .wb_ack_o(wb_ack_o),
        .accel_in_data_bus(accel_in_data_bus),
        .accel_channel_select(accel_channel_select),
        .accel_write_en(accel_write_en),
        .accel_write_xy_sel(accel_write_xy_sel),
        .accel_start_op(accel_start_op),
        .accel_binary_result(accel_binary_result),
        .accel_valid(accel_valid)
    );

    // Instantiate the accelerator core.
    // (Make sure the port names below match your accelerator's definition.)
    stochastic_multiplier_adder_top_serial #(
        .BIN_WIDTH(BIN_WIDTH),
        .ST_LENGTH(ST_LENGTH),
        .NUM_CHANNELS(NUM_CHANNELS)
    ) U_accel_core (
        .in_data_bus(accel_in_data_bus),
        .channel_select(accel_channel_select),
        .write_en(accel_write_en),
        .write_xy_sel(accel_write_xy_sel),
        .start_op(accel_start_op),
        .binary_result(accel_binary_result),
        .valid(accel_valid),
        .clk(clk),
        .reset(reset)
    );

    // Route lower 8 bits of the accelerator result to debug LEDs.
    assign debug_leds = accel_binary_result[7:0];

endmodule
```

Listing 5: Wishbone-Integrated Top-Level Accelerator

A.6 Vivado Constraints

```
## Clock and Reset Constraints
# Replace PIN_CLK and PIN_RST with valid pin names for your board.
set_property PACKAGE_PIN PIN_CLK [get_ports {clk}]
set_property IOSTANDARD LVCMOS33 [get_ports {clk}]
create_clock -name clk -period 10 [get_ports {clk}]

set_property PACKAGE_PIN PIN_RST [get_ports {reset}]
set_property IOSTANDARD LVCMOS33 [get_ports {reset}]

## Serial Interface for in_data_bus (32-bit vector)
# Explicitly assign each bit a valid pin. Replace PIN_D0 ... PIN_D31 with
    valid names.
set_property PACKAGE_PIN PIN_D0 [get_ports {in_data_bus[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[0]}]

set_property PACKAGE_PIN PIN_D1 [get_ports {in_data_bus[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[1]}]

set_property PACKAGE_PIN PIN_D2 [get_ports {in_data_bus[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[2]}]

set_property PACKAGE_PIN PIN_D3 [get_ports {in_data_bus[3]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[3]}]

set_property PACKAGE_PIN PIN_D4 [get_ports {in_data_bus[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[4]}]

set_property PACKAGE_PIN PIN_D5 [get_ports {in_data_bus[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[5]}]

set_property PACKAGE_PIN PIN_D6 [get_ports {in_data_bus[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[6]}]

set_property PACKAGE_PIN PIN_D7 [get_ports {in_data_bus[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[7]}]

set_property PACKAGE_PIN PIN_D8 [get_ports {in_data_bus[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[8]}]

set_property PACKAGE_PIN PIN_D9 [get_ports {in_data_bus[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[9]}]

set_property PACKAGE_PIN PIN_D10 [get_ports {in_data_bus[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[10]}]

set_property PACKAGE_PIN PIN_D11 [get_ports {in_data_bus[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[11]}]

set_property PACKAGE_PIN PIN_D12 [get_ports {in_data_bus[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[12]}]

set_property PACKAGE_PIN PIN_D13 [get_ports {in_data_bus[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[13]}]

set_property PACKAGE_PIN PIN_D14 [get_ports {in_data_bus[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[14]}]

set_property PACKAGE_PIN PIN_D15 [get_ports {in_data_bus[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[15]}]

set_property PACKAGE_PIN PIN_D16 [get_ports {in_data_bus[16]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[16]}]

set_property PACKAGE_PIN PIN_D17 [get_ports {in_data_bus[17]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[17]}]

set_property PACKAGE_PIN PIN_D18 [get_ports {in_data_bus[18]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[18]}]

set_property PACKAGE_PIN PIN_D19 [get_ports {in_data_bus[19]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[19]}]

set_property PACKAGE_PIN PIN_D20 [get_ports {in_data_bus[20]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[20]}]

set_property PACKAGE_PIN PIN_D21 [get_ports {in_data_bus[21]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[21]}]

set_property PACKAGE_PIN PIN_D22 [get_ports {in_data_bus[22]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[22]}]

set_property PACKAGE_PIN PIN_D23 [get_ports {in_data_bus[23]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[23]}]

set_property PACKAGE_PIN PIN_D24 [get_ports {in_data_bus[24]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[24]}]

set_property PACKAGE_PIN PIN_D25 [get_ports {in_data_bus[25]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[25]}]

set_property PACKAGE_PIN PIN_D26 [get_ports {in_data_bus[26]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[26]}]

set_property PACKAGE_PIN PIN_D27 [get_ports {in_data_bus[27]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[27]}]

set_property PACKAGE_PIN PIN_D28 [get_ports {in_data_bus[28]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[28]}]

set_property PACKAGE_PIN PIN_D29 [get_ports {in_data_bus[29]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[29]}]

set_property PACKAGE_PIN PIN_D30 [get_ports {in_data_bus[30]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[30]}]

set_property PACKAGE_PIN PIN_D31 [get_ports {in_data_bus[31]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data_bus[31]}]

## channel_select (5-bit vector)
set_property PACKAGE_PIN PIN_CS0 [get_ports {channel_select[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {channel_select[0]}]
set_property PACKAGE_PIN PIN_CS1 [get_ports {channel_select[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {channel_select[1]}]
set_property PACKAGE_PIN PIN_CS2 [get_ports {channel_select[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {channel_select[2]}]
set_property PACKAGE_PIN PIN_CS3 [get_ports {channel_select[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {channel_select[3]}]
set_property PACKAGE_PIN PIN_CS4 [get_ports {channel_select[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {channel_select[4]}]

## write_en
set_property PACKAGE_PIN PIN_WE [get_ports write_en]
set_property IOSTANDARD LVCMOS33 [get_ports write_en]

## write_xy_sel
set_property PACKAGE_PIN PIN_WX [get_ports write_xy_sel]
set_property IOSTANDARD LVCMOS33 [get_ports write_xy_sel]

## start_op
set_property PACKAGE_PIN PIN_SO [get_ports start_op]
set_property IOSTANDARD LVCMOS33 [get_ports start_op]

## binary_result (32-bit output)
# Assign explicit pins for each bit (replace PIN_BR0 .. PIN_BR31 with valid
    names)
set_property PACKAGE_PIN PIN_BR0  [get_ports {binary_result[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[0]}]
set_property PACKAGE_PIN PIN_BR1  [get_ports {binary_result[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[1]}]
set_property PACKAGE_PIN PIN_BR2  [get_ports {binary_result[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[2]}]
set_property PACKAGE_PIN PIN_BR3  [get_ports {binary_result[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[3]}]
set_property PACKAGE_PIN PIN_BR4  [get_ports {binary_result[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[4]}]
set_property PACKAGE_PIN PIN_BR5  [get_ports {binary_result[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[5]}]
set_property PACKAGE_PIN PIN_BR6  [get_ports {binary_result[6]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[6]}]
set_property PACKAGE_PIN PIN_BR7  [get_ports {binary_result[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[7]}]
set_property PACKAGE_PIN PIN_BR8  [get_ports {binary_result[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[8]}]
set_property PACKAGE_PIN PIN_BR9  [get_ports {binary_result[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[9]}]
set_property PACKAGE_PIN PIN_BR10 [get_ports {binary_result[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[10]}]
set_property PACKAGE_PIN PIN_BR11 [get_ports {binary_result[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[11]}]
set_property PACKAGE_PIN PIN_BR12 [get_ports {binary_result[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[12]}]
set_property PACKAGE_PIN PIN_BR13 [get_ports {binary_result[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[13]}]
set_property PACKAGE_PIN PIN_BR14 [get_ports {binary_result[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[14]}]
set_property PACKAGE_PIN PIN_BR15 [get_ports {binary_result[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[15]}]
set_property PACKAGE_PIN PIN_BR16 [get_ports {binary_result[16]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[16]}]
set_property PACKAGE_PIN PIN_BR17 [get_ports {binary_result[17]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[17]}]
set_property PACKAGE_PIN PIN_BR18 [get_ports {binary_result[18]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[18]}]
set_property PACKAGE_PIN PIN_BR19 [get_ports {binary_result[19]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[19]}]
set_property PACKAGE_PIN PIN_BR20 [get_ports {binary_result[20]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[20]}]
set_property PACKAGE_PIN PIN_BR21 [get_ports {binary_result[21]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[21]}]
set_property PACKAGE_PIN PIN_BR22 [get_ports {binary_result[22]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[22]}]
set_property PACKAGE_PIN PIN_BR23 [get_ports {binary_result[23]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[23]}]
set_property PACKAGE_PIN PIN_BR24 [get_ports {binary_result[24]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[24]}]
set_property PACKAGE_PIN PIN_BR25 [get_ports {binary_result[25]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[25]}]
set_property PACKAGE_PIN PIN_BR26 [get_ports {binary_result[26]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[26]}]
set_property PACKAGE_PIN PIN_BR27 [get_ports {binary_result[27]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[27]}]
set_property PACKAGE_PIN PIN_BR28 [get_ports {binary_result[28]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[28]}]
set_property PACKAGE_PIN PIN_BR29 [get_ports {binary_result[29]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[29]}]
set_property PACKAGE_PIN PIN_BR30 [get_ports {binary_result[30]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[30]}]
set_property PACKAGE_PIN PIN_BR31 [get_ports {binary_result[31]}]
set_property IOSTANDARD LVCMOS33 [get_ports {binary_result[31]}]

## valid
set_property PACKAGE_PIN PIN_VLD [get_ports valid]
set_property IOSTANDARD LVCMOS33 [get_ports valid]

## Optional: Override Clock Dedicated Routing (if necessary)
#set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clk_IBUF]
```

Listing 6: Clock and Pin Constraints for SCAU

A.7 MAC benchmark accelerator source code and constraints file

```
`timescale 1ns / 1ps
module Maced #(
    parameter WIDTH = 16,
    parameter PIPELINE_DEPTH = 16
)(
    input clk,
    input reset,
    input start,
    input [WIDTH-1:0] a,
    input [WIDTH-1:0] b,
    input [WIDTH*2-1:0] c,
    output reg [WIDTH*2-1:0] result,
    output finish
);

    // Intermediate registers for the computed MAC result and valid signal.
    reg [WIDTH*2-1:0] computed;
    reg valid_computed;

    // Pipeline registers for the computed result and valid flag.
    reg [WIDTH*2-1:0] result_pipe [0:PIPELINE_DEPTH-1];
    reg [PIPELINE_DEPTH-1:0] valid_pipe;

    // The finish signal is high when the last stage holds a valid result.
    assign finish = valid_pipe[PIPELINE_DEPTH-1];

    integer i;

    // First always block: Compute the MAC result.
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            computed       <= 0;
            valid_computed <= 0;
        end else begin
            computed       <= a * b + c;
            valid_computed <= start;
        end
    end

    // Second always block: Shift the computed result through the pipeline.
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            for (i = 0; i < PIPELINE_DEPTH; i = i + 1) begin
                result_pipe[i] <= 0;
                valid_pipe[i]  <= 0;
            end
            result <= 0;
        end else begin
            // Load the computed result into stage 0.
            result_pipe[0] <= computed;
            valid_pipe[0]  <= valid_computed;
            // Shift pipeline registers.
            for (i = 1; i < PIPELINE_DEPTH; i = i + 1) begin
                result_pipe[i] <= result_pipe[i-1];
                valid_pipe[i]  <= valid_pipe[i-1];
            end
            // Output the result when the last stage is valid.
            result <= result_pipe[PIPELINE_DEPTH-1];
        end
    end

```

```
endmodule
##############################################################################
## XDC Constraints File for Maced Module
##
## This sample file assigns example FPGA pins to the Maced module's
## ports. Update the PACKAGE_PIN assignments to match your board's
## pinout. The IOSTANDARD is set to LVCMOS33 for all signals.
##############################################################################

## Clock input
set_property PACKAGE_PIN W5 [get_ports {clk}]
set_property IOSTANDARD LVCMOS33 [get_ports {clk}]
create_clock -period 10.000 -name clk -waveform {0 5} [get_ports {clk}]

## Reset input (active high)
set_property PACKAGE_PIN V4 [get_ports {reset}]
set_property IOSTANDARD LVCMOS33 [get_ports {reset}]

## Start signal
set_property PACKAGE_PIN U4 [get_ports {start}]
set_property IOSTANDARD LVCMOS33 [get_ports {start}]

## Input bus "a" (16-bit)
set_property PACKAGE_PIN A1  [get_ports {a[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[0]}]
set_property PACKAGE_PIN A2  [get_ports {a[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[1]}]
set_property PACKAGE_PIN A3  [get_ports {a[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[2]}]
set_property PACKAGE_PIN A4  [get_ports {a[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[3]}]
set_property PACKAGE_PIN A5  [get_ports {a[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[4]}]
set_property PACKAGE_PIN A6  [get_ports {a[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[5]}]
set_property PACKAGE_PIN A7  [get_ports {a[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[6]}]
set_property PACKAGE_PIN A8  [get_ports {a[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[7]}]
set_property PACKAGE_PIN A9  [get_ports {a[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[8]}]
set_property PACKAGE_PIN A10 [get_ports {a[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[9]}]
set_property PACKAGE_PIN A11 [get_ports {a[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[10]}]
set_property PACKAGE_PIN A12 [get_ports {a[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[11]}]
set_property PACKAGE_PIN A13 [get_ports {a[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[12]}]
set_property PACKAGE_PIN A14 [get_ports {a[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[13]}]
set_property PACKAGE_PIN A15 [get_ports {a[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[14]}]
set_property PACKAGE_PIN A16 [get_ports {a[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[15]}]

## Input bus "b" (16-bit)
set_property PACKAGE_PIN B1  [get_ports {b[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[0]}]
set_property PACKAGE_PIN B2  [get_ports {b[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[1]}]
set_property PACKAGE_PIN B3  [get_ports {b[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[2]}]
```

```
set_property PACKAGE_PIN B4  [get_ports {b[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[3]}]
set_property PACKAGE_PIN B5  [get_ports {b[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[4]}]
set_property PACKAGE_PIN B6  [get_ports {b[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[5]}]
set_property PACKAGE_PIN B7  [get_ports {b[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[6]}]
set_property PACKAGE_PIN B8  [get_ports {b[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[7]}]
set_property PACKAGE_PIN B9  [get_ports {b[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[8]}]
set_property PACKAGE_PIN B10 [get_ports {b[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[9]}]
set_property PACKAGE_PIN B11 [get_ports {b[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[10]}]
set_property PACKAGE_PIN B12 [get_ports {b[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[11]}]
set_property PACKAGE_PIN B13 [get_ports {b[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[12]}]
set_property PACKAGE_PIN B14 [get_ports {b[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[13]}]
set_property PACKAGE_PIN B15 [get_ports {b[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[14]}]
set_property PACKAGE_PIN B16 [get_ports {b[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[15]}]

## Input bus "c" (32-bit)
set_property PACKAGE_PIN C1  [get_ports {c[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[0]}]
set_property PACKAGE_PIN C2  [get_ports {c[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[1]}]
set_property PACKAGE_PIN C3  [get_ports {c[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[2]}]
set_property PACKAGE_PIN C4  [get_ports {c[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[3]}]
set_property PACKAGE_PIN C5  [get_ports {c[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[4]}]
set_property PACKAGE_PIN C6  [get_ports {c[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[5]}]
set_property PACKAGE_PIN C7  [get_ports {c[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[6]}]
set_property PACKAGE_PIN C8  [get_ports {c[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[7]}]
set_property PACKAGE_PIN C9  [get_ports {c[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[8]}]
set_property PACKAGE_PIN C10 [get_ports {c[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[9]}]
set_property PACKAGE_PIN C11 [get_ports {c[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[10]}]
set_property PACKAGE_PIN C12 [get_ports {c[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[11]}]
set_property PACKAGE_PIN C13 [get_ports {c[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[12]}]
set_property PACKAGE_PIN C14 [get_ports {c[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[13]}]
set_property PACKAGE_PIN C15 [get_ports {c[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[14]}]
set_property PACKAGE_PIN C16 [get_ports {c[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[15]}]
set_property PACKAGE_PIN C17 [get_ports {c[16]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[16]}]
set_property PACKAGE_PIN C18 [get_ports {c[17]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {c[17]}]
set_property PACKAGE_PIN C19 [get_ports {c[18]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[18]}]
set_property PACKAGE_PIN C20 [get_ports {c[19]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[19]}]
set_property PACKAGE_PIN C21 [get_ports {c[20]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[20]}]
set_property PACKAGE_PIN C22 [get_ports {c[21]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[21]}]
set_property PACKAGE_PIN C23 [get_ports {c[22]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[22]}]
set_property PACKAGE_PIN C24 [get_ports {c[23]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[23]}]
set_property PACKAGE_PIN C25 [get_ports {c[24]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[24]}]
set_property PACKAGE_PIN C26 [get_ports {c[25]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[25]}]
set_property PACKAGE_PIN C27 [get_ports {c[26]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[26]}]
set_property PACKAGE_PIN C28 [get_ports {c[27]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[27]}]
set_property PACKAGE_PIN C29 [get_ports {c[28]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[28]}]
set_property PACKAGE_PIN C30 [get_ports {c[29]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[29]}]
set_property PACKAGE_PIN C31 [get_ports {c[30]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[30]}]
set_property PACKAGE_PIN C32 [get_ports {c[31]}]
set_property IOSTANDARD LVCMOS33 [get_ports {c[31]}]

## Output bus "result" (32-bit)
set_property PACKAGE_PIN D1  [get_ports {result[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[0]}]
set_property PACKAGE_PIN D2  [get_ports {result[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[1]}]
set_property PACKAGE_PIN D3  [get_ports {result[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[2]}]
set_property PACKAGE_PIN D4  [get_ports {result[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[3]}]
set_property PACKAGE_PIN D5  [get_ports {result[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[4]}]
set_property PACKAGE_PIN D6  [get_ports {result[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[5]}]
set_property PACKAGE_PIN D7  [get_ports {result[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[6]}]
set_property PACKAGE_PIN D8  [get_ports {result[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[7]}]
set_property PACKAGE_PIN D9  [get_ports {result[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[8]}]
set_property PACKAGE_PIN D10 [get_ports {result[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[9]}]
set_property PACKAGE_PIN D11 [get_ports {result[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[10]}]
set_property PACKAGE_PIN D12 [get_ports {result[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[11]}]
set_property PACKAGE_PIN D13 [get_ports {result[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[12]}]
set_property PACKAGE_PIN D14 [get_ports {result[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[13]}]
set_property PACKAGE_PIN D15 [get_ports {result[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[14]}]
set_property PACKAGE_PIN D16 [get_ports {result[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[15]}]
```

```
set_property PACKAGE_PIN D17 [get_ports {result[16]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[16]}]
set_property PACKAGE_PIN D18 [get_ports {result[17]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[17]}]
set_property PACKAGE_PIN D19 [get_ports {result[18]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[18]}]
set_property PACKAGE_PIN D20 [get_ports {result[19]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[19]}]
set_property PACKAGE_PIN D21 [get_ports {result[20]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[20]}]
set_property PACKAGE_PIN D22 [get_ports {result[21]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[21]}]
set_property PACKAGE_PIN D23 [get_ports {result[22]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[22]}]
set_property PACKAGE_PIN D24 [get_ports {result[23]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[23]}]
set_property PACKAGE_PIN D25 [get_ports {result[24]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[24]}]
set_property PACKAGE_PIN D26 [get_ports {result[25]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[25]}]
set_property PACKAGE_PIN D27 [get_ports {result[26]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[26]}]
set_property PACKAGE_PIN D28 [get_ports {result[27]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[27]}]
set_property PACKAGE_PIN D29 [get_ports {result[28]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[28]}]
set_property PACKAGE_PIN D30 [get_ports {result[29]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[29]}]
set_property PACKAGE_PIN D31 [get_ports {result[30]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[30]}]
set_property PACKAGE_PIN D32 [get_ports {result[31]}]
set_property IOSTANDARD LVCMOS33 [get_ports {result[31]}]

## Output signal "finish"
set_property PACKAGE_PIN F1 [get_ports {finish}]
set_property IOSTANDARD LVCMOS33 [get_ports {finish}]
```

Listing 7: Full source code for the benchmark MAC desighn with constraints file included

A.8 PlatformIO C-Code for Running Calculations through the SCAU

```c
#include <stdint.h>
#include <stdio.h>

// Known working peripheral
#define GPIO_LEDS     0x80001404

// Your accelerator data register
#define ACCEL_DATA    0x80003200
#define ACCEL_CTRL    0x80003204
#define ACCEL_RESULT  0x80003208
#define ACCEL_STATUS  0x8000320C

#define WRITE_REG(addr, val) (*(volatile uint32_t *)(addr) = (val))
#define READ_REG(addr)       (*(volatile uint32_t *)(addr))

void delay(volatile int count) {
    while (count-- > 0) {
        __asm__ volatile ("nop");
    }
}
```

```c

int main() {
    uint32_t status = 0;
    uint32_t result = 0;

    printf("=== Address Reachability Test ===\n");

    // 1. Blink GPIO LEDs to confirm MMIO works
    printf("[TEST] Writing to GPIO LEDs (0x80001404)...\n");
    for (int i = 0; i < 4; i++) {
        WRITE_REG(GPIO_LEDS, 0xAA);
        delay(100000);
        WRITE_REG(GPIO_LEDS, 0x55);
        delay(100000);
    }
    printf("[PASS] GPIO LEDs responded.\n");

    // 2. Write X = 0x05 to ACCEL_DATA (bit13 = 0)
    printf("[TEST] Writing X = 0x05 to ACCEL_DATA (0x80003200)...\n");
    WRITE_REG(ACCEL_DATA, 0x000000AA); // 0xAA = 0x05 << 8 (channel 0,
        xy_select = 0)

    // 3. Write Y = 0x07 to ACCEL_DATA (bit13 = 1)
    printf("[TEST] Writing Y = 0x07 to ACCEL_DATA (bit13=1)...\n");
    WRITE_REG(ACCEL_DATA, 0x0000205);

    // 4. Start the accelerator
    printf("[TEST] Writing 0x01 to ACCEL_CTRL (0x80003204) to start...\n");
    WRITE_REG(ACCEL_CTRL, 0x00000001);

    // 5. Wait for 'done' flag in status
    printf("[INFO] Polling ACCEL_STATUS (0x8000320C)...\n");
    int timeout = 0;
    while (((status = READ_REG(ACCEL_STATUS)) & 0x00000001) == 0 && timeout <
         100000) {
        timeout++;
    }

    if (status & 0x1) {
        printf("[PASS] Accelerator reports DONE flag.\n");
    } else {
        printf("[FAIL] Accelerator did not respond (timeout waiting for DONE)
            .\n");
    }

    // 6. Read result
    result = READ_REG(ACCEL_RESULT);
    printf("[INFO] ACCEL_RESULT (0x80003208) = 0x%08X\n", result);

    printf("=== Test Complete ===\n");



    return 0;
}
```

Listing 8: C code for running calculations through the SCAU on RISC-V

Appendix B: Test bench Code

B.1 Stochastic MAC Operation Testbench

```
`timescale 1ns/1ps

module tb_binary_to_stochastic_complete_bitstream;

  // DUT parameters
  parameter BIN_WIDTH = 8;
  parameter ST_LENGTH = 256;    // Total bit stream length to collect
  parameter PARALLEL  = 4;
  parameter [7:0] SEED = 8'hA5;

  // Test bench parameter: fixed binary input value for generating the bit
      stream.
  parameter TEST_INPUT = 255;

  // Tolerance for validation (fraction of total bits)
  parameter real TOLERANCE = 0.05;

  // Derived parameter: number of clock cycles required to collect ST_LENGTH
      bits.
  localparam NUM_CYCLES = (ST_LENGTH + PARALLEL - 1) / PARALLEL;

  // Signal declarations.
  reg clk;
  reg reset;
  reg clk_en;
  reg [BIN_WIDTH-1:0] binary_input;
  wire [PARALLEL-1:0] stochastic_output;

  // Instantiate the DUT.
  binary_to_stochastic #(
    .BIN_WIDTH(BIN_WIDTH),
    .ST_LENGTH(ST_LENGTH),
    .PARALLEL(PARALLEL),
    .SEED(SEED)
  ) dut (
    .clk(clk),
    .reset(reset),
    .clk_en(clk_en),
    .binary_input(binary_input),
    .stochastic_output(stochastic_output)
  );

  // Clock generation: 10 ns period clock (100 MHz).
  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end

  // Dump simulation waveforms for visual analysis (e.g., with GTKWave).
  initial begin
    $dumpfile("tb_binary_to_stochastic_complete.vcd");
    $dumpvars(0, tb_binary_to_stochastic_complete_bitstream);
  end

  // Variables for bit stream collection and validation.
  reg [ST_LENGTH-1:0] complete_bit_stream;
  integer cycle_count;  // Loop variable for clock cycles.
  integer idx;          // Index for appending bits in the bit stream.
```

```
57    integer bit_idx;      // Loop variable for iterating over
          complete_bit_stream.
58    integer j;            // Loop variable for inner loop.
59    integer ones_count;
60    real expected_prob;
61    real expected_count;
62    real diff;
63
64    // File handle for output logging.
65    integer bitstream_file;
66
67    // Main test bench process.
68    initial begin
69      // Open a file (CSV format) for logging results.
70      bitstream_file = $fopen("complete_bitstream_results.csv", "w");
71      if (bitstream_file == 0) begin
72        $display("ERROR: Could not open complete_bitstream_results.csv for
              writing.");
73        $finish;
74      end
75      $fdisplay(bitstream_file, "TEST_INPUT,ST_LENGTH,Collected_Ones,
            Expected_Ones,Result");
76
77      // Initialize signals and clear the bit stream.
78      reset = 1;
79      clk_en = 0;
80      binary_input = {BIN_WIDTH{1'b0}};
81      complete_bit_stream = {ST_LENGTH{1'b0}};
82      idx = 0;
83      ones_count = 0;
84      #20;  // Hold reset.
85
86      // Release reset, enable clock, and apply the fixed test input.
87      reset = 0;
88      clk_en = 1;
89      binary_input = TEST_INPUT;
90
91      // Allow a few cycles for stabilization.
92      repeat (5) @(posedge clk);
93
94      // Collect the complete bit stream over NUM_CYCLES clock cycles.
95      // Each cycle, copy PARALLEL bits from stochastic_output.
96      for (cycle_count = 0; cycle_count < NUM_CYCLES; cycle_count = cycle_count
            + 1) begin
97        @(posedge clk);
98        for (j = 0; j < PARALLEL; j = j + 1) begin
99          complete_bit_stream[idx + j] = stochastic_output[j];
100       end
101       idx = idx + PARALLEL;
102     end
103
104     // Display and log the complete bit stream.
105     $display("Complete Bit Stream for TEST_INPUT = %0d:", TEST_INPUT);
106     $display("%b", complete_bit_stream);
107     $fdisplay(bitstream_file, "Complete Bit Stream for TEST_INPUT = %0d",
            TEST_INPUT);
108     $fdisplay(bitstream_file, "%b", complete_bit_stream);
109
110     // Validate the bit stream by counting the number of ones.
111     ones_count = 0;
112     for (bit_idx = 0; bit_idx < ST_LENGTH; bit_idx = bit_idx + 1) begin
113       if (complete_bit_stream[bit_idx])
114         ones_count = ones_count + 1;
```

```
    end

    // Calculate the expected probability and expected count.
    // Expected probability = (TEST_INPUT + 1) / 256.
    expected_prob = (TEST_INPUT + 1.0) / 256.0;
    expected_count = expected_prob * ST_LENGTH;

    // Compute the absolute difference between collected and expected ones.
    diff = (ones_count > expected_count) ? (ones_count - expected_count)
                                         : (expected_count - ones_count);

    // Validate the bit stream within the defined tolerance.
    if (diff <= TOLERANCE * ST_LENGTH) begin
      $display("PASS: Bit stream validation passed.");
      $display("Collected ones = %0d, Expected ones = %.2f", ones_count,
          expected_count-1);
      $fdisplay(bitstream_file, "%0d,%0d,%0d,%.2f,PASS", TEST_INPUT,
          ST_LENGTH, ones_count, expected_count);
    end else begin
      $display("FAIL: Bit stream validation failed.");
      $display("Collected ones = %0d, Expected ones = %.2f", ones_count,
          expected_count-1);
      $fdisplay(bitstream_file, "%0d,%0d,%0d,%.2f,FAIL", TEST_INPUT,
          ST_LENGTH, ones_count, expected_count);
    end

    $fclose(bitstream_file);
    #50;
    $finish;
  end

endmodule
```

Listing 9: Testbench for Stochastic Multiplier and Adder

B.3 Wishbone Interface Testbench

```
module tb_stochastic_multiplier_adder_top_serial;
    // Parameter definitions
    parameter BIN_WIDTH    = 8;
    parameter ST_LENGTH    = 256;
    parameter NUM_CHANNELS = 1;
    parameter NUM_TESTS    = 1000;

    // DUT interface signals
    reg  [31:0] in_data_bus;
    localparam CH_SEL_WIDTH = (NUM_CHANNELS <= 1) ? 1 : $clog2(NUM_CHANNELS);
    reg  [CH_SEL_WIDTH-1:0] channel_select;
    reg         write_en;
    reg         write_xy_sel; // 0 => write X, 1 => write Y
    reg         start_op;
    wire [31:0] binary_result;
    wire        valid;

    // Clock and reset signals
    reg clk;
    reg reset;

    // Simulation variables
    integer timeout_counter;
    integer i, ch, x_val, y_val, golden_sum;
```

```
    real    percent_error;
    time    calc_start_time_t, calc_end_time_t;
    real    calc_time_r, total_calc_time;
    integer calc_count;

    // Performance measurements
    time testbench_start_time, testbench_end_time;
    integer total_calculations;
    real total_time_ns, avg_pipeline_latency, throughput_ns_per_input;

    // Error tracking
    real total_percent_error, mean_percent_error;

    // CSV file descriptor for logging results
    integer data_file;

    // Instantiate DUT
    stochastic_multiplier_adder_top_serial #(
        .BIN_WIDTH(BIN_WIDTH),
        .ST_LENGTH(ST_LENGTH),
        .NUM_CHANNELS(NUM_CHANNELS)
    ) DUT (
        .in_data_bus(in_data_bus),
        .channel_select(channel_select),
        .write_en(write_en),
        .write_xy_sel(write_xy_sel),
        .start_op(start_op),
        .binary_result(binary_result),
        .valid(valid),
        .clk(clk),
        .reset(reset)
    );

    // Clock generation: 10 ns period (100 MHz)
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    // Main test procedure
    initial begin
        //================== FILE SETUP ==================
        data_file = $fopen("results.csv", "w");
        if (data_file == 0) begin
            $display("ERROR: Could not open results.csv for writing.");
            $finish;
        end
        $fdisplay(data_file, "Test#,Golden Result,DUT Result,Percent Error,
            Calc Time (ns)");
        //================================================

        total_calc_time     = 0.0;
        calc_count          = 0;
        total_percent_error = 0.0;
        testbench_start_time = $realtime;

        // Reset
        reset         = 1;
        write_en      = 0;
        start_op      = 0;
        channel_select= 0;
        write_xy_sel  = 0;
        in_data_bus   = 0;
```

```
87          repeat (2) @(posedge clk);
88          reset = 0;
89          repeat (2) @(posedge clk);
90
91          // Main test loop
92          for (i = 0; i < NUM_TESTS; i = i + 1) begin
93              golden_sum = 0;
94
95              // Write random 8-bit values to all channels
96              for (ch = 0; ch < NUM_CHANNELS; ch = ch + 1) begin
97                  x_val = $random & 8'hFF;
98                  y_val = $random & 8'hFF;
99
100                 // Write X value
101                 @(posedge clk);
102                 channel_select = ch[CH_SEL_WIDTH-1:0];
103                 write_xy_sel   = 0;
104                 in_data_bus    = x_val;
105                 write_en       = 1;
106                 @(posedge clk);
107                 write_en       = 0;
108
109                 // Write Y value
110                 @(posedge clk);
111                 channel_select = ch[CH_SEL_WIDTH-1:0];
112                 write_xy_sel   = 1;
113                 in_data_bus    = y_val;
114                 write_en       = 1;
115                 @(posedge clk);
116                 write_en       = 0;
117
118                 golden_sum = golden_sum + (x_val * y_val);
119             end
120
121             // Issue start pulse
122             @(posedge clk);
123             start_op = 1;
124             @(posedge clk);
125             start_op = 0;
126             @(posedge clk);
127             calc_start_time_t = $realtime;
128
129             // Wait for result
130             timeout_counter = 0;
131             while (!valid && timeout_counter < 100000) begin
132                 @(posedge clk);
133                 timeout_counter = timeout_counter + 1;
134             end
135
136             if (!valid) begin
137                 $display("ERROR: Timeout waiting for valid signal at test %0d
                        .", i);
138                 $finish;
139             end
140
141             // Capture timing and compute error
142             calc_end_time_t = $realtime;
143             calc_time_r     = calc_end_time_t - calc_start_time_t;
144             total_calc_time = total_calc_time + calc_time_r;
145             calc_count      = calc_count + 1;
146
147             if (golden_sum == 0)
148                 percent_error = (binary_result == 0) ? 0.0 : 100.0;
```

```
            else if (binary_result >= golden_sum)
                percent_error = 100.0 * (binary_result - golden_sum) /
                    golden_sum;
            else
                percent_error = 100.0 * (golden_sum - binary_result) /
                    golden_sum;

            // Add to total percent error for MPE
            total_percent_error = total_percent_error + percent_error;

            // Log result
            $fdisplay(data_file, "%0d,%0d,%0d,%.2f,%.2f",
                      i, golden_sum, binary_result, percent_error,
                          calc_time_r);

            $display("Test %3d: GOLDEN=%d, RESULT=%d, ERR=%.2f%%, CALC_TIME
                =%.1f ns",
                     i, golden_sum, binary_result, percent_error, calc_time_r
                         );

            @(posedge clk);
        end

        $fclose(data_file);
        testbench_end_time  = $realtime;
        total_calculations  = NUM_TESTS * NUM_CHANNELS;
        total_time_ns       = testbench_end_time - testbench_start_time;
        avg_pipeline_latency = (calc_count > 0) ? total_calc_time /
            calc_count : 0.0;
        throughput_ns_per_input = (total_calculations > 0) ? total_calc_time
            / total_calculations : 0.0;

        // ====================== OUTPUT =====================
        $display("\n--- Performance Results ---");
        $display("Total Inputs Processed: %0d", total_calculations);
        $display("Average Pipeline Latency: %.2f ns", avg_pipeline_latency);
        $display("Total Time Taken: %.2f ns", total_time_ns);
        $display("Throughput: %.2f ns per input", throughput_ns_per_input);

        // Mean Percent Error Calculation
        mean_percent_error = total_percent_error / NUM_TESTS;
        $display("\n--- Error Metrics ---");
        $display("Mean Percent Error (MPE): %.4f%%", mean_percent_error);

        $display("All tests completed.");
        $finish;
    end
endmodule
```

Listing 10: SystemVerilog Testbench for Wishbone Interface

B.2 test bench the multiplier unit of SCAU

```

`timescale 1ns/1ps

module sc_multiplier_adder_counter_stochastic_tb;

  //------------------------------------------------------------------------
  // Parameters (must match the DUT)
  //------------------------------------------------------------------------
```

```
  localparam integer BIN_WIDTH    = 8;
  localparam integer ST_LENGTH    = 256;
  localparam integer NUM_CHANNELS = 1;
  localparam integer PARALLEL     = 1;
  // SHIFT_SCALE = (1 << (2*BIN_WIDTH))/(ST_LENGTH)
  localparam integer SHIFT_SCALE  = (1 << (2 * BIN_WIDTH)) / ST_LENGTH;

  // Probability threshold for stochastic bit generation (0-100)
  localparam integer THRESHOLD = 75;

  //------------------------------------------------------------------------
  // Signal Declarations
  //------------------------------------------------------------------------
  reg clk;
  reg reset;
  reg clk_en;
  reg [NUM_CHANNELS*PARALLEL-1:0] stochastic_x;
  reg [NUM_CHANNELS*PARALLEL-1:0] stochastic_y;
  wire [31:0] result;
  wire valid;

  //------------------------------------------------------------------------
  // DUT Instantiation
  //------------------------------------------------------------------------
  sc_multiplier_adder_counter #(
    .BIN_WIDTH(BIN_WIDTH),
    .ST_LENGTH(ST_LENGTH),
    .NUM_CHANNELS(NUM_CHANNELS)
  ) dut (
    .clk(clk),
    .reset(reset),
    .clk_en(clk_en),
    .stochastic_x(stochastic_x),
    .stochastic_y(stochastic_y),
    .result(result),
    .valid(valid)
  );

  //------------------------------------------------------------------------
  // Clock Generation: 10 ns period (100 MHz clock)
  //------------------------------------------------------------------------
  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end

  //------------------------------------------------------------------------
  // Test Variables
  //------------------------------------------------------------------------
  integer cycle;
  integer bit;
  integer cycle_sum;     // Sum for each cycle (number of ANDs that are 1)
  integer sum_expected;  // Total expected sum over ST_LENGTH cycles
  reg [NUM_CHANNELS*PARALLEL-1:0] temp_x;
  reg [NUM_CHANNELS*PARALLEL-1:0] temp_y;

  //------------------------------------------------------------------------
  // Stochastic Bitstream Test
  //------------------------------------------------------------------------
  initial begin
    // Initialize variables
    sum_expected   = 0;
    stochastic_x   = 0;
```

```
    stochastic_y    = 0;
    clk_en          = 0;
    reset           = 1;

    // Apply reset for one clock cycle
    @(posedge clk);
    reset = 0;
    clk_en = 1;

    // Run for ST_LENGTH cycles
    for (cycle = 0; cycle < ST_LENGTH; cycle = cycle + 1) begin
      // Generate stochastic bitstreams for x and y
      for (bit = 0; bit < NUM_CHANNELS*PARALLEL; bit = bit + 1) begin
        temp_x[bit] = (($urandom % 100) < THRESHOLD) ? 1'b1 : 1'b0;
        temp_y[bit] = (($urandom % 100) < THRESHOLD) ? 1'b1 : 1'b0;
      end

      // Drive the inputs
      stochastic_x <= temp_x;
      stochastic_y <= temp_y;

      // Wait one clock cycle for the inputs to be sampled
      @(posedge clk);

      // Calculate the number of 1's from the bitwise AND of temp_x and
          temp_y
      cycle_sum = 0;
      for (bit = 0; bit < NUM_CHANNELS*PARALLEL; bit = bit + 1) begin
        cycle_sum = cycle_sum + (temp_x[bit] & temp_y[bit]);
      end

      // Accumulate the cycle sum
      sum_expected = sum_expected + cycle_sum;
    end

    // Extra clock cycle so the DUT can latch the final result and assert
        valid.
    @(posedge clk);

    // Wait for the DUT to assert valid
    wait (valid == 1);

    // The expected scaled result is the accumulated sum multiplied by
        SHIFT_SCALE.
    if (result == sum_expected * SHIFT_SCALE)
      $display("Stochastic test PASSED: result = %0d, expected = %0d", result
          , sum_expected * SHIFT_SCALE);
    else
      $display("Stochastic test FAILED: result = %0d, expected = %0d", result
          , sum_expected * SHIFT_SCALE);

    $finish;
  end

endmodule
```

Listing 11: test bench of the multiplier unit and stochastic to binary unit of the SCAU

B.3 test bench of benchmark MAC accelerator

```
1
```

```
`timescale 1ns / 1ps

module pipelined_mac_tb;

  parameter WIDTH          = 16;
  parameter NUM_TESTS      = 1000;
  parameter PIPELINE_DEPTH = 16;

  reg  clk;
  reg  reset;
  reg  start;
  reg  [WIDTH-1:0]    a;
  reg  [WIDTH-1:0]    b;
  reg  [2*WIDTH-1:0]  c;
  wire [2*WIDTH-1:0]  result;
  wire                finish;

  reg [WIDTH-1:0]     a_array [0:NUM_TESTS-1];
  reg [WIDTH-1:0]     b_array [0:NUM_TESTS-1];
  reg [2*WIDTH-1:0]   c_array [0:NUM_TESTS-1];
  reg [2*WIDTH-1:0]   golden_array [0:NUM_TESTS-1];

  reg [31:0] test_index_queue [0:NUM_TESTS-1];
  integer    queue_head = 0;
  integer    queue_tail = 0;
  integer    current_test_index;

  integer i;
  integer difference;
  real    error_percent;
  real    temp_golden;
  real    temp_diff;

  real    start_time;
  real    first_finish_time;
  real    end_time;
  integer first_finish_captured = 0;
  integer total_inputs_processed = 0;
  real    ns_per_input;
  real    throughput;

  reg finish_delayed;

  Maced #(
    .WIDTH(WIDTH),
    .PIPELINE_DEPTH(PIPELINE_DEPTH)
  ) uut (
    .clk(clk),
    .reset(reset),
    .start(start),
    .a(a),
    .b(b),
    .c(c),
    .result(result),
    .finish(finish)
  );

  // Clock generation: 10 ns period
  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end

```

```
  // Initialize test vectors
  initial begin
    reset = 1;
    start = 0;
    a = 0;
    b = 0;
    c = 0;
    for (i = 0; i < NUM_TESTS; i = i + 1) begin
      a_array[i]      = i + 1;
      b_array[i]      = (i + 1) * 2;
      c_array[i]      = (i + 1) * 3;
      golden_array[i] = (a_array[i] * b_array[i]) + c_array[i];
    end
    #20;
    reset = 0;
  end

  // Drive inputs every cycle and compute performance
  initial begin
    #30;
    @(posedge clk);
    test_index_queue[queue_head] = -1;
    queue_head = queue_head + 1;
    @(posedge clk);
    start_time = $realtime;

    for (i = 0; i < NUM_TESTS; i = i + 1) begin
      @(posedge clk);
      start <= 1;
      a <= a_array[i];
      b <= b_array[i];
      c <= c_array[i];
      test_index_queue[queue_head] = i;
      queue_head = queue_head + 1;
      total_inputs_processed = total_inputs_processed + 1;
    end

    // Stop input after final input
    @(posedge clk);
    start <= 0;

    // Wait for pipeline to flush
    repeat (PIPELINE_DEPTH + 10) @(posedge clk);

    end_time = $realtime;

    ns_per_input = (end_time - start_time) / total_inputs_processed;
    throughput = total_inputs_processed / (end_time - start_time);

    $display("\n--- Performance Results ---");
    $display("Total Inputs Processed: %0d", total_inputs_processed);
    if (first_finish_captured)
      $display("Pipeline Latency: %.2f ns", first_finish_time - start_time);
    else
      $display("Pipeline Latency: <not detected>");
    $display("Total Time Taken: %.2f ns", end_time - start_time);
    $display("Average Delay: %.4f ns per input", ns_per_input);
    $display("Throughput: %.4f inputs/ns (%.2f inputs/sec)", throughput,
        throughput*1e9);
    $display("-----------------------------\n");

    $finish;
  end
```

```
127
128   // Output checking and latency capture
129   always @(posedge clk) begin
130     finish_delayed <= finish;
131     if (finish && !finish_delayed) begin
132       if (!first_finish_captured) begin
133         first_finish_time = $realtime;
134         first_finish_captured = 1;
135       end
136
137       current_test_index = test_index_queue[queue_tail];
138       queue_tail = queue_tail + 1;
139
140       if (current_test_index != -1) begin
141         difference   = $signed(result) - $signed(golden_array[
                current_test_index]);
142         temp_golden  = $itor($signed(golden_array[current_test_index]));
143         temp_diff    = $itor(difference);
144         error_percent = (temp_golden != 0.0) ? (temp_diff * 100.0) /
                temp_golden : 0.0;
145
146         $display("Test %0d: GOLDEN=%0d, RESULT=%0d, ERR=%.2f%%, TIME=%.2f ns"
                ,
147                  current_test_index,
148                  golden_array[current_test_index],
149                  result,
150                  error_percent,
151                  $realtime);
152       end
153     end
154   end
155
156 endmodule
```

Listing 12: Test bench for overall SCAU desighn testing speed and chip usage

Appendix C: Graphs and figures

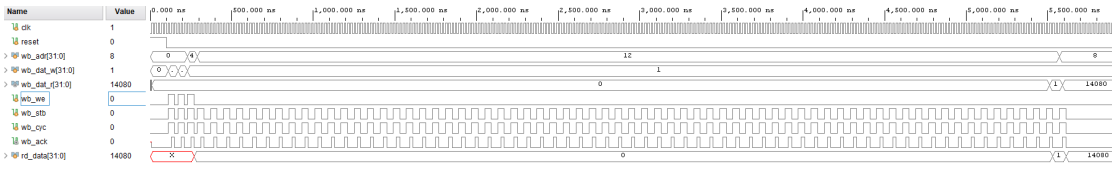

Figure 1: *Example Wishbone-Bus Transaction Waveform for SCAU Accelerator*

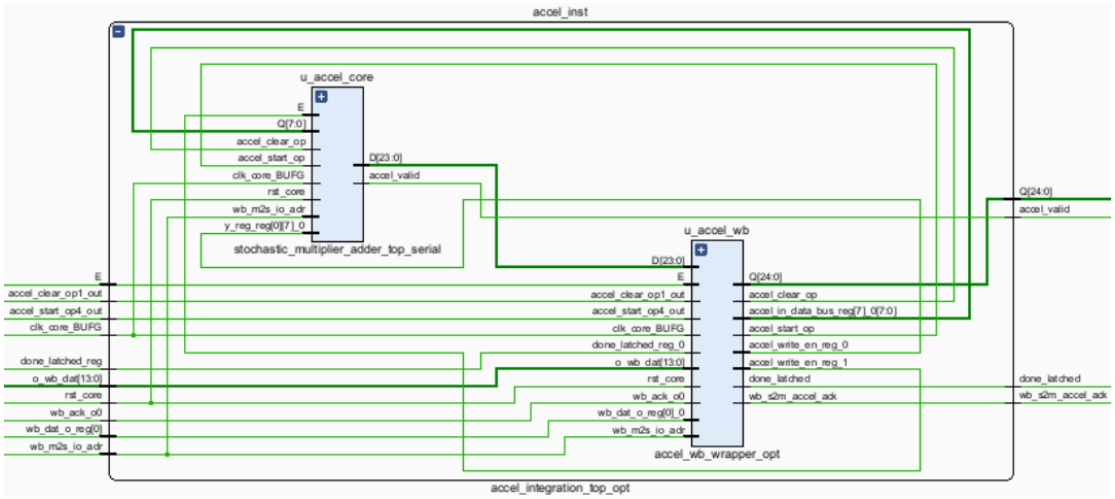

Figure 2: *Example of the Risc-V integrated SCAU chip on vivado*