Degree Project in Embedded Systems

Second cycle, 30 credits

Hardware Accelerator for a Data-driven Nonlinear State Estimation Method (DANSE) Based on Silago Architecture

ZIXUN YAN

Hardware Accelerator for a Data-driven Nonlinear State Estimation Method (DANSE) Based on Silago Architecture

ZIXUN YAN

Master's Programme, Embedded Systems, 120 credits
Date: June 24, 2026

Supervisor: Paul Delestrac
Examiner: Ahmed Hemani
School of Electrical Engineering and Computer Science
Swedish title: Hårdvaruaccelerator för den datadrivna icke-linjära tillståndsskattningsmetoden DANSE baserad på Silago-arkitekturen

© 2026 Zixun Yan

Abstract

Data-driven Nonlinear State Estimation (DANSE) combines Bayesian estimation with recurrent neural networks to perform state prediction without requiring explicit system models. Although DANSE provides strong estimation performance, its computational structure poses challenges for deployment on resource-constrained hardware platforms.

This thesis investigates the deployment of DANSE-derived computational kernels on the SiLago coarse-grained reconfigurable architecture (CGRA), which is built upon the Dynamic Reconfigurable Resource Array (DRRA). The focus is placed on the matrix-vector multiplication and bias-addition operations that dominate the computational workload of the DANSE prior network. To enable efficient execution on DRRA, the original floating-point model is reformulated using fixed-point arithmetic, and a Q8.8 numerical representation is selected through range and accuracy analysis.

A deployment methodology is proposed that transforms DANSE computations into reusable hardware applications. The extracted kernels are unified into two representative matrix-vector computation platforms with dimensions 32×32 and 32×3. To support these workloads, the existing DRRA processing unit is extended with a Q8.8 arithmetic resource that implements fixed-point addition, multiplication, and multiply-accumulate operations. The resulting kernels are implemented using PASM and validated through a multi-level verification flow consisting of functional verification, SST architectural simulation, and RTL simulation.

Experimental results show that the selected Q8.8 representation preserves the numerical behavior of the original DANSE model while enabling efficient hardware execution. Bitwise-identical outputs are obtained across Model0, SST, and RTL verification levels for both implemented kernels, demonstrating the correctness and consistency of the proposed hardware design across multiple abstraction levels. The proposed workflow demonstrates a complete deployment path from DANSE-derived neural-network computations to verified hardware implementations on the SiLago platform and provides a foundation for future acceleration of recurrent neural-network workloads on DRRA-based architectures.

Keywords

Data-Driven State Estimation, Fixed-Point Arithmetic, Hardware Acceleration, DRRA, SiLago

Sammanfattning

Datadriven icke-linjär tillståndsskattning (Data-driven Nonlinear State Estimation, DANSE) kombinerar Bayesiansk skattning med rekurrenta neurala nätverk för att utföra tillståndsprediktion utan att kräva explicita systemmodeller. Trots att DANSE uppvisar hög skattningsnoggrannhet innebär dess beräkningsstruktur betydande utmaningar vid implementering på hårdvaruplattformar med begränsade resurser.

I detta examensarbete undersöks implementeringen av DANSE-härledda beräkningskärnor på den grovkornigt rekonfigurerbara arkitekturen (CGRA) SiLago, som bygger på Dynamic Reconfigurable Resource Array (DRRA). Fokus ligger på de matris-vektor-multiplikationer och biasadditioner som dominerar beräkningsbelastningen i DANSE-modellens priornätverk. För att möjliggöra effektiv exekvering på DRRA omformuleras den ursprungliga flyttalsmodellen till fixpunktsaritmetik, och representationen Q8.8 väljs genom analys av numeriskt intervall och noggrannhet.

En metodik föreslås för att omvandla DANSE-beräkningar till återanvändbara hårdvaruapplikationer. De extraherade kärnorna generaliseras till två representativa plattformar för matris-vektor-beräkning med dimensionerna 32×32 respektive 32×3. För att stödja dessa arbetsbelastningar utökas den befintliga DRRA-processorenheten med en Q8.8-baserad aritmetisk resurs som implementerar fixpunktsaddition, multiplikation och multiply-accumulate-operationer. De resulterande kärnorna implementeras med hjälp av PASM och verifieras genom ett flernivåförfarande bestående av funktionell verifiering, SST-baserad arkitektursimulering och RTL-simulering.

Resultaten visar att den valda Q8.8-representationen bevarar det numeriska beteendet hos den ursprungliga DANSE-modellen samtidigt som den möjliggör effektiv hårdvaruexekvering. Bitidentiska resultat erhålls mellan Model0, SST- och RTL-verifieringsnivåerna för båda implementerade kärnor, vilket bekräftar korrekthet och konsistens hos den föreslagna hårdvarudesignen över flera abstraktionsnivåer. Det föreslagna arbetsflödet demonstrerar en komplett implementeringskedja från DANSE-härledda neurala nätverksberäkningar till verifierade hårdvaruimplementationer på SiLago-plattformen och utgör en grund för framtida acceleration av rekurrenta neurala nätverk på DRRA-baserade arkitekturer.

Nyckelord

Datadriven tillståndsskattning, Fixpunktsaritmetik, CGRA, DRRA, SiLago, Hårdvaruacceleration, Rekurrenta neurala nätverk

Acknowledgments

This thesis marks the end of a challenging but rewarding journey. Looking back, the most difficult part was not the implementation itself, but the amount of knowledge that had to be acquired before any meaningful work could begin. My previous background was primarily in embedded systems and hardware-related development, while this project requires me to learn concepts from neural networks and chaotic-system prediction, areas in which I had very limited prior experience. At the same time, the work required me to break down an algorithmic framework and transform it into a hardware-friendly deployment flow, before eventually implementing and verifying it at the hardware level. Looking back, I often felt that I was moving across multiple disciplines throughout a single project, working with technologies ranging from Python and machine-learning frameworks to assembly programming and hardware description languages. The learning curve was steep and the preparation effort was substantial, but it also gave me valuable insight into how hardware accelerators are developed in practice, from algorithm understanding to hardware realization.

I would like to express my sincere gratitude to Professor Saikat Chatterjee and Professor Ahmed Hemani. Professor Chatterjee, from the Department of Information Science and Technology, was willing to discuss research ideas with me and provided the foundation through his work on the DANSE framework. Professor Hemani, as my examiner, gave me the freedom to explore a direction that matched my interests and ultimately made this thesis topic possible by supporting the idea of deploying DANSE on the SiLago architecture.

I would also like to thank my supervisors, Yu Yang and Paul Delestrac, who provided the majority of the guidance throughout this work. Their patience, technical expertise, and continuous support were invaluable. They were always willing to answer questions, arrange discussions, and help resolve difficulties whenever they arose. Even during the transition between supervisors, they ensured that the project continued smoothly without losing direction or momentum.

Finally, I would like to express my appreciation for the open and supportive academic environment in Sweden. The freedom to explore ideas independently, make mistakes, and learn through experimentation has been an important part of my education. This environment allowed me to approach problems creatively and develop both technical skills and research experience throughout my studies.

List of acronyms and abbreviations

AGU	Address Generation Unit
CGRA	Coarse-Grained Reconfigurable Architecture
DANSE	Data-driven Nonlinear State Estimation
DPU	Data Processing Unit
DRRA	Dynamic Reconfigurable Resource Array
FSM	Finite State Machine
GRU	Gated Recurrent Unit
ISA	Instruction Set Architecture
MAC	Multiply-Accumulate
MSE	Mean Square Error
MVM	Matrix-Vector Multiplication
NMSE	Normalized Mean Square Error
PASM	Parallel Assembly
RF	Register File
RNN	Recurrent Neural Network
RTL	Register Transfer Level
SMNR	Signal-to-Measurement-Noise Ratio
SST	Structural Simulation Toolkit
SWB	Switch Box

Contents

1 Introduction **xii**
- 1.1 Background . . . xii
- 1.2 Problem . . . xiii
  - 1.2.1 Original Problem Definition . . . xiii
  - 1.2.2 Scientific and Engineering Issues . . . xiii
- 1.3 Purpose . . . xiv
- 1.4 Goals . . . xiv
- 1.5 Research Methodology . . . xiv
- 1.6 Delimitations . . . xv

2 Background **xvii**
- 2.1 DANSE-Based State Estimation . . . xvii
  - 2.1.1 State Estimation . . . xvii
  - 2.1.2 DANSE Model . . . xviii
  - 2.1.3 GRU Neural Network . . . xix
- 2.2 SiLago Architecture . . . xx
  - 2.2.1 Coarse-Grained Reconfigurable Architectures . . . xx
  - 2.2.2 SiLago Design Paradigm . . . xxi
  - 2.2.3 Programming and Mapping Flow . . . xxii
  - 2.2.4 Suitability for Neural Network Computation . . . xxii
- 2.3 Related Work . . . xxiii
  - 2.3.1 Hardware Acceleration for Neural Networks . . . xxiii
  - 2.3.2 FPGA-based Neural Network Acceleration . . . xxiii
  - 2.3.3 ASIC-based Neural Network Acceleration . . . xxiv
  - 2.3.4 CGRA-based Neural Network Acceleration . . . xxiv
  - 2.3.5 Research Gap . . . xxv

3 Method **xxvii**
- 3.1 Data Collection and Dataset Generation . . . xxvii

3.1.1 State-Space Model xxvii
3.1.2 Dynamical Systems xxviii
3.1.3 Dataset Generation xxviii
3.1.4 Training and Testing Setup xxix
3.2 Application Analysis and Partitioning xxix
3.2.1 Floating-Point Baseline xxix
3.2.2 Parameter Extraction xxix
3.2.3 Workload Analysis and Model Decomposition xxx
3.2.4 Computational Kernel Extraction xxxi
3.3 Numerical Representation Selection xxxi
3.3.1 Quantization Procedure xxxii
3.3.2 Fixed-Point Representation Design xxxii
3.3.3 Evaluation Method xxxii
3.4 Hardware Mapping Methodology xxxiii
3.4.1 Application Selection xxxiii
3.4.2 Memory Layout for Input and Output Buffers xxxiv
3.4.3 Model0 Reference Design xxxiv
3.4.4 Resource Binding Strategy xxxv
3.4.5 Arithmetic Adaptation xxxvi
3.4.6 Verification Methodology xxxvi

4 Software Implementation and Quantization xxxviii
4.1 Pretrained Model and Evaluation Data xxxviii
4.2 Hardware-Oriented Decomposition of DANSE xxxix
4.2.1 Forward-Pass Decomposition xxxix
4.2.2 Posterior Simplification xl
4.3 Fixed-Point Quantization for Hardware Mapping xli
4.3.1 Q*m*.*n* Representation xli
4.3.2 Fixed-Point Arithmetic Reformulation xlii
4.3.3 Integer Range Analysis xlii
4.3.4 Precision Exploration xliii

5 Hardware Mapping of DANSE on SiLago xliv
5.1 DRRA Architecture and Resource Organization xlv
5.2 Instruction Set Extension and PASM Operations xlvii
5.3 SST-Based Q8.8 Resource Implementation xlviii
5.3.1 Structure of the Q8.8 DPU Resource xlviii
5.3.2 Data Conversion and Fixed-Point Arithmetic xlix
5.3.3 Multiply-Accumulate Operation xlix

5.3.4 Operation Handler Structure l
5.3.5 Event Scheduling and Execution Behavior l
5.4 RTL-Based Resource Implementation li
5.4.1 RTL Resource Structure li
5.4.2 Fixed-Point Arithmetic Modules lii
5.4.3 Multiply-Accumulate Datapath lii
5.4.4 Consistency with the SST Model liii
5.5 PASM-Based Mapping of Matrix-Vector Kernels liii
5.5.1 Initial Three-Cell Mapping liii
5.5.2 Optimized Single-Cell Mapping liv
5.5.3 Input Buffer Layout liv
5.5.4 Bulk Data Transfer lv
5.5.5 Narrow Dataflow inside the Computation Cell lv
5.5.6 Scheduling Difference Between the Two Kernels . . . lvi
5.5.7 Routing and Constraint-Based Scheduling lvi
5.5.8 Summary lvii

6 Results and Analysis **lx**
6.1 Software Quantization Results lx
6.1.1 Integer Bit-Width Evaluation lx
6.1.2 End-to-End State Estimation Results lxi
6.1.3 Q-format Selection lxii
6.2 Functional Verification lxii
6.2.1 Verification Flow lxii
6.2.2 Verification Results lxiv
6.3 Architectural Evaluation lxv
6.3.1 Execution Cycle Analysis lxv
6.3.2 Resource Utilization Analysis lxvi
6.3.3 Comparison with Software Execution lxvii
6.3.4 Discussion lxviii

7 Conclusions and Future Work **lxx**
7.1 Conclusions lxx
7.2 Limitations lxxi
7.2.1 Hardware Acceleration Scope lxxi
7.2.2 Hardware Evaluation Metrics lxxii
7.2.3 Scalability Evaluation lxxii
7.2.4 Architectural Exploration lxxii
7.3 Future Work lxxii

7.3.1 Deployment of Complete Neural-Network Workloads . lxxiii
7.3.2 Automated Mapping and Code Generation lxxiii
7.3.3 Extended Architectural Exploration lxxiii
7.3.4 Physical Implementation Evaluation lxxiii
7.3.5 Support for Additional Neural-Network Operations . . lxxiii
7.4 Reflections lxxiv

References **lxxiv**

Chapter 1

Introduction

1.1 Background

Recent advances in machine learning have led to the widespread adoption of neural-network-based models in applications such as autonomous systems, robotics, industrial monitoring, and Internet of Things (IoT) devices. While these models can achieve high prediction accuracy, their deployment on resource-constrained hardware platforms remains challenging due to the computational complexity and memory requirements of neural-network inference.

One common approach to improving deployment efficiency is numerical quantization, where floating-point computations are replaced by lower-precision representations. Quantization reduces arithmetic complexity, memory footprint, and hardware cost, making it an important technique for implementing machine-learning workloads on dedicated hardware accelerators.

At the same time, coarse-grained reconfigurable architectures (CGRAs) have emerged as an attractive alternative to conventional processor-based execution. By exposing computation, communication, and storage resources explicitly, CGRAs provide opportunities for application-specific optimization while maintaining a degree of flexibility. The SiLago [1] framework is one such architecture, consisting of a Dynamic Reconfigurable Resource Array (DRRA) and a composable instruction set that enables efficient mapping of streaming applications onto reconfigurable hardware fabrics.

The work presented in this thesis is motivated by the deployment of computations originating from the Data-driven Nonlinear State Estimation (DANSE) [2] framework. DANSE combines recurrent neural networks

with Bayesian estimation techniques to perform nonlinear state estimation and prediction. Analysis of the DANSE model shows that matrix-vector multiplication and bias-addition operations constitute dominant computational kernels within the prior network. These kernels therefore serve as representative workloads for investigating hardware-oriented deployment strategies on the SiLago DRRA architecture.

1.2 Problem

1.2.1 Original Problem Definition

The original objective of this project was to investigate the feasibility of deploying computationally intensive components of the DANSE framework onto the SiLago architecture. While DANSE provides accurate nonlinear state estimation, its implementation relies on floating-point neural-network computations that are not directly suitable for efficient hardware deployment.

Preliminary analysis revealed that the dominant operations within the prior network can be represented as matrix-vector multiplication and bias-addition kernels. Consequently, the project evolved from a full-model deployment problem into the investigation of how representative neural-network computations can be transformed, mapped, and verified on the DRRA architecture.

1.2.2 Scientific and Engineering Issues

The work addresses the following research questions:

1. Which numerical representation can preserve the computational behavior of the target neural-network kernels while remaining suitable for hardware implementation?

2. How can the DRRA architecture be extended to efficiently support fixed-point neural-network computations?

3. How can representative neural-network kernels be mapped and verified across multiple hardware abstraction levels within the SiLago framework?

4. What tradeoffs exist between alternative resource-allocation and mapping strategies on the DRRA architecture?

1.3 Purpose

The purpose of this thesis is to develop and evaluate a deployment methodology for quantized neural-network computations on the SiLago DRRA architecture.

The proposed methodology covers the complete implementation flow from numerical quantization and architectural extension to PASM-based mapping, multi-level verification, and architectural evaluation. By using representative matrix-vector multiplication kernels derived from DANSE as a case study, the work aims to demonstrate how neural-network-inspired computations can be transformed into hardware-oriented implementations suitable for deployment on coarse-grained reconfigurable architectures.

1.4 Goals

To achieve the stated purpose, the thesis pursues the following goals:

1. Identify a hardware-friendly numerical representation capable of maintaining acceptable computational accuracy.
2. Develop architectural support for fixed-point neural-network computations within the DRRA framework.
3. Investigate mapping strategies for deploying representative matrix-vector multiplication kernels on the SiLago architecture.
4. Establish a verification methodology spanning software, architectural, and RTL abstraction levels.
5. Evaluate the tradeoffs between alternative mapping and resource-allocation strategies.

1.5 Research Methodology

The research follows a hardware-oriented design methodology consisting of five main stages.

In the beginning, representative computational kernels are extracted from the target workload and analyzed from both numerical and architectural perspectives. Then, different numerical representations are evaluated to identify suitable fixed-point formats. The DRRA architecture is also extended

with fixed-point arithmetic support. Next, the resulting kernels are mapped onto the architecture using PASM and verified across multiple abstraction levels, including software reference models, architectural simulation, and RTL implementation. Finally, architectural tradeoffs are evaluated through performance and resource-utilization analysis.

1.6 Delimitations

This thesis focuses on matrix-vector multiplication and bias-addition kernels extracted from the DANSE framework. The complete DANSE model, including training procedures, recurrent-network control logic, and activation-function acceleration, is outside the scope of this work.

The evaluation focuses on numerical correctness, architectural functionality, and mapping-related tradeoffs. Physical implementation aspects such as silicon area, post-layout timing closure, and power consumption are not considered and are left for future work.

Chapter 2

Background

This chapter introduces the background required to understand the proposed fixed-point implementation of the DANSE model. It first reviews nonlinear state estimation and the role of recurrent neural networks in data-driven state estimation. It then describes fixed-point arithmetic, Q8.8 representation, and the SiLago/DRRA architecture. Finally, related work on DANSE, GRU acceleration, fixed-point neural networks, and hardware mapping approaches is discussed.

2.1 DANSE-Based State Estimation

State estimation aims to infer the hidden states of a dynamic system from noisy observations and has been widely applied in control, signal processing, and tracking applications. Traditional approaches are often model-based and rely on accurate knowledge of system dynamics. Recent advances in machine learning have enabled data-driven alternatives that reduce dependence on explicit system models. This section introduces the fundamental concepts of state estimation and presents the Deep Auto-Regressive Neural State Estimator (DANSE), which forms the algorithmic foundation of this thesis. The recurrent neural network architecture employed by DANSE is also briefly reviewed.

2.1.1 State Estimation

State estimation is a fundamental problem in many engineering applications, including control systems, robotics, autonomous navigation, and signal processing. The goal is to infer the hidden internal state of a system from a

sequence of observations, which are often corrupted by noise. In mathematical terms, a dynamical system can be described as:

$$x_k = f(x_{k-1}, w_k) \tag{2.1}$$
$$y_k = h(x_k, v_k) \tag{2.2}$$

where x_k denotes the system state at time step k, y_k represents the observations, and w_k and v_k denote process and measurement noise, respectively. The functions $f(\cdot)$ and $h(\cdot)$ describe the system dynamics and observation model.

In linear systems with Gaussian noise, optimal state estimation can be achieved using the Kalman filter. However, in many real-world scenarios, the system dynamics are nonlinear and the noise distributions are unknown or non-Gaussian. In such cases, classical approaches such as the Extended Kalman Filter (EKF) or Unscented Kalman Filter (UKF) may suffer from reduced accuracy or increased computational complexity.

To address these challenges, data-driven approaches have been proposed, where the system dynamics are learned directly from data rather than being explicitly modeled. These approaches leverage machine learning techniques to approximate the underlying state transition and observation functions, making them suitable for complex nonlinear systems where analytical models are unavailable.

2.1.2 DANSE Model

Data-driven Nonlinear State Estimation (DANSE) is a framework for estimating the hidden state of a nonlinear dynamical system from a sequence of noisy observations. Unlike classical model-based filters, DANSE does not require the complete analytical form of the underlying system dynamics. Instead, it uses a data-driven recurrent model to learn temporal dependencies from observation sequences.

Figure 2.1 shows the simplified inference flow used to describe DANSE in this thesis. The model first receives the input sequence $y_{1:t-1}$, which contains the observations available before the current time step. This sequence is processed by the prior update module, implemented using a recurrent neural network such as a GRU. The prior update extracts temporal information from past observations and generates information used to predict the current state distribution.

At time step t, the current observation input y_t is introduced into the posterior update module. The posterior update combines the information from the prior update with the current observation to refine the state estimate. The final output of the inference pipeline is the state estimate x_t.

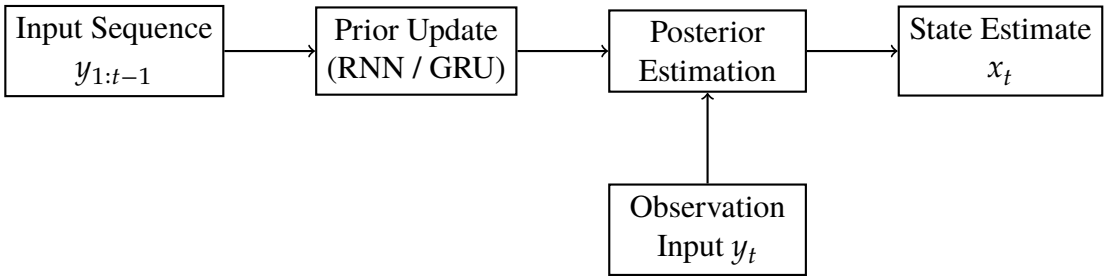

Figure 2.1: Simplified DANSE inference pipeline

2.1.3 GRU Neural Network

The Gated Recurrent Unit (GRU) is a type of recurrent neural network designed to model sequential data while mitigating issues such as vanishing gradients. GRU introduces gating mechanisms that control the flow of information across time steps, allowing the network to capture long-term dependencies efficiently.

At each time step k, the GRU updates its hidden state h_k using the following equations:

$$z_k = \sigma(W_z y_k + U_z h_{k-1} + b_z) \tag{2.3}$$
$$r_k = \sigma(W_r y_k + U_r h_{k-1} + b_r) \tag{2.4}$$
$$\tilde{h}_k = \tanh(W_h y_k + U_h(r_k \odot h_{k-1}) + b_h) \tag{2.5}$$
$$h_k = (1 - z_k) \odot h_{k-1} + z_k \odot \tilde{h}_k \tag{2.6}$$

where:

- z_k is the update gate,
- r_k is the reset gate,
- $\tilde{h}_k$ is the candidate hidden state,
- $\sigma(\cdot)$ denotes the sigmoid activation function,

- $\odot$ represents element-wise multiplication.

From a computational perspective, GRU operations consist primarily of:

- Matrix-vector multiplications ($W \cdot x$ and $U \cdot h$)
- Vector additions (bias terms)
- Element-wise operations (multiplication and addition)
- Nonlinear activation functions (sigmoid and hyperbolic tangent)

Among these operations, matrix-vector multiplication dominates the computational cost and is therefore a key target for optimization. In typical implementations, these operations take the form:

$$[m \times n] \times [n \times 1] + [m]$$

which aligns with the structure of neuron updates in neural networks.

In the context of this thesis, the GRU component of the DANSE model is analyzed and decomposed into hardware-friendly operations. This enables the transformation of floating-point computations into fixed-point representations and facilitates efficient mapping onto hardware architectures such as SiLago.

2.2 SiLago Architecture

Efficient execution of neural-network-based state estimation algorithms requires a hardware platform capable of exploiting computational parallelism while maintaining flexibility for different workloads. This thesis targets the SiLago coarse-grained reconfigurable architecture (CGRA), which provides a scalable and programmable platform for data-intensive computations. This section introduces the architectural concepts underlying CGRAs and presents the organization of the SiLago system and its Dynamically Reconfigurable Resource Array (DRRA).

2.2.1 Coarse-Grained Reconfigurable Architectures

Coarse-Grained Reconfigurable Architectures (CGRAs) have emerged as an efficient solution for accelerating computationally intensive applications, particularly in embedded and signal processing domains. CGRAs provide a balance between the high efficiency of application-specific integrated circuits

(ASICs) and the flexibility of field-programmable gate arrays (FPGAs). They consist of an array of processing elements interconnected through a configurable network, enabling parallel execution of operations.

Compared to fine-grained architectures, CGRAs operate on word-level data and support higher computational throughput with lower control overhead. This makes them particularly suitable for applications characterized by regular dataflow patterns, such as linear algebra operations and neural network inference. As a result, CGRAs are increasingly adopted for accelerating machine learning and digital signal processing workloads.

2.2.2 SiLago Design Paradigm

SiLago is a VLSI design framework based on the concept of composable hardware blocks, often referred to as "Silicon Lego". The framework emphasizes composability, predictability, and scalability by constructing large systems from pre-designed and pre-characterized blocks.

A key feature of SiLago is that hardware blocks can be composed by abutment, without requiring full logic or physical synthesis during system integration. This enables rapid design space exploration and predictable cost estimation at higher levels of abstraction. The design flow maintains consistency across functional behavior, timing, and resource usage, making it suitable for large-scale reconfigurable systems.

Within the SiLago framework, the Dynamically Reconfigurable Resource Array (DRRA) serves as the computational fabric and can be classified as a Coarse-Grained Reconfigurable Architecture (CGRA). The DRRA consists of an array of processing elements interconnected through configurable communication structures, enabling spatial execution of dataflow-oriented applications.

Each DRRA cell includes several key components:

- A Data Path Unit (DPU) responsible for arithmetic and logical operations,
- A Register File (RF) providing local storage,
- A Sequencer (SEQ) acting as a local control unit,
- Switch Boxes (SB) enabling communication with neighboring cells.

The DPU operates as the main computational unit, while the RF stores intermediate data required for computation. The SEQ dynamically

configures the behavior of the DPU and communication paths at runtime. Communication between cells is achieved through a localized interconnection network, often implemented using a sliding window mechanism, allowing each cell to exchange data with a limited set of neighboring cells.

2.2.3 Programming and Mapping Flow

The SiLago toolchain provides a complete workflow for mapping applications onto the DRRA-based hardware platform.

The process begins with the identification and reformulation of computational applications suitable for execution on the DRRA architecture. Based on the selected application, a PASM (Parallel Assembly) description is manually constructed to define computation, communication, and memory-access behavior.

The Vesyla tool is then used to compile the PASM program and generate the artifacts required for simulation and verification. Resource allocation, scheduling, and timing validation are performed according to the constraints specified in the PASM description.

The PASM program describes computation at the resource level. PASM instructions specify how processing elements, memory units, and interconnects are configured and coordinated during execution. Unlike traditional assembly languages, PASM does not represent a sequential instruction stream but instead encodes parallel operations across distributed hardware resources.

Following compilation, the toolchain generates simulation models at different abstraction levels. A cycle-accurate simulation is performed using an SST-based model for functional and timing verification. In addition, a register-transfer level (RTL) model is generated, enabling detailed hardware simulation using tools such as ModelSim or QuestaSim.

This unified workflow allows efficient design, verification, and evaluation of applications on the SiLago platform, bridging the gap between high-level algorithm design and hardware realization.

2.2.4 Suitability for Neural Network Computation

The DRRA-based architecture is particularly suitable for neural network inference workloads due to its support for parallel and streaming computation. Operations such as matrix-vector multiplication and vector updates can be mapped efficiently onto the array of processing elements.

Furthermore, the distributed memory and localized communication structure enable efficient data reuse and reduce memory access overhead. These characteristics make the SiLago platform a promising candidate for accelerating the computational kernels present in recurrent neural networks and state estimation models.

2.3 Related Work

2.3.1 Hardware Acceleration for Neural Networks

The increasing complexity of machine-learning models has created a growing demand for efficient hardware acceleration. Neural-network workloads are dominated by matrix multiplication, accumulation, and memory-access operations, making them computationally expensive when executed on general-purpose processors. Consequently, numerous hardware platforms have been proposed to improve computational efficiency while reducing energy consumption.

Mittal [3] provided a comprehensive review of FPGA-based neural-network accelerators and showed that customized hardware architectures can significantly improve throughput and energy efficiency compared with conventional CPU implementations. Similarly, Wang *et al.* [4] surveyed modern deep-learning accelerators and highlighted the importance of numerical quantization, memory optimization, and specialized arithmetic units in achieving efficient inference performance.

These studies demonstrate that hardware acceleration has become an essential component of modern machine-learning deployment. However, existing surveys primarily focus on accelerator architectures and performance optimization, while comparatively less attention is paid to deployment methodologies that bridge the gap between software models and hardware implementations.

2.3.2 FPGA-based Neural Network Acceleration

Field-Programmable Gate Arrays (FPGAs) have become one of the most widely adopted platforms for neural-network acceleration due to their balance between flexibility and performance.

Zhang *et al.* [5] proposed an FPGA-based accelerator for convolutional neural networks and demonstrated that exploiting loop unrolling and processing-element parallelism can significantly improve computational

throughput. Their work established matrix multiplication as one of the dominant computational kernels in neural-network inference and motivated subsequent research on customized MAC architectures.

Venieris and Bouganis [6] investigated automated toolflows for mapping neural networks onto FPGA platforms. Their work demonstrated that compiler-assisted optimization and dataflow scheduling can substantially improve resource utilization and performance. Similarly, Liu [7] developed a dedicated matrix multiplication accelerator and showed that customized datapaths can efficiently support MAC-intensive workloads.

Although FPGA accelerators provide considerable flexibility, resource allocation is typically fixed after synthesis. Consequently, architectural exploration and dynamic resource management remain limited compared with more explicitly reconfigurable architectures.

2.3.3 ASIC-based Neural Network Acceleration

Application-Specific Integrated Circuits (ASICs) provide an alternative approach to neural-network acceleration by optimizing hardware structures for specific workloads.

Wang *et al.* [8] investigated structured-weight-matrix accelerators for deep neural networks and demonstrated that exploiting matrix regularity can significantly reduce computational complexity and storage requirements. Their results showed that specialized hardware structures can achieve high computational efficiency while reducing implementation cost.

For recurrent neural-network workloads, Nurvitadhi *et al.* [9] surveyed hardware acceleration techniques for RNNs and highlighted matrix-vector multiplication as the dominant operation in GRU- and LSTM-based inference. Their study demonstrated the effectiveness of specialized arithmetic architectures for recurrent-network acceleration.

While ASIC accelerators typically provide superior energy efficiency and performance, they generally sacrifice flexibility. Supporting new workloads often requires redesigning the hardware architecture, making rapid architectural exploration more difficult.

2.3.4 CGRA-based Neural Network Acceleration

Coarse-Grained Reconfigurable Architectures (CGRAs) have emerged as a promising compromise between the flexibility of FPGAs and the efficiency of ASICs. By exposing arithmetic, storage, and communication resources at

the architectural level, CGRAs enable application-specific optimization while maintaining programmability.

Kim *et al.* [10] proposed a stream-processing CGRA architecture for object-inference applications and demonstrated that spatially distributed computation can improve throughput while preserving flexibility. Their work showed that machine-learning workloads can benefit from explicit communication and computation scheduling.

More recently, Guo *et al.* [11] introduced ML-CGRA, an integrated compilation framework for machine-learning acceleration on CGRA platforms. Their work demonstrated that compiler-assisted mapping can effectively translate machine-learning workloads onto reconfigurable arrays. Similarly, Schönberger *et al.* [12] investigated highly parameterized CGRA architectures and showed that architectural flexibility enables efficient adaptation to different machine-learning workloads.

These studies have significantly advanced CGRA-based machine-learning acceleration. However, most existing work focuses on architecture design, scheduling strategies, or compiler development rather than deployment methodologies that span multiple abstraction levels.

2.3.5 Research Gap

The reviewed literature demonstrates substantial progress in hardware acceleration across FPGA, ASIC, and CGRA platforms. FPGA studies primarily focus on accelerator optimization and throughput improvement, while ASIC research emphasizes efficiency and workload specialization. Existing CGRA work has largely concentrated on architecture design and compiler-assisted mapping.

However, limited work has investigated a complete deployment workflow that combines numerical quantization, architectural extension, low-level mapping, and verification within a CGRA environment. In particular, existing studies rarely address the following aspects simultaneously:

- Selection of hardware-oriented fixed-point representations for neural-network computations;
- Extension of arithmetic resources to support quantized execution;
- Low-level mapping of neural-network kernels onto CGRA fabrics;
- Verification across software, architectural simulation, and RTL abstraction levels.

To address these gaps, this thesis proposes and evaluates a deployment methodology for quantized neural-network computations on the SiLago DRRA architecture. The proposed workflow integrates numerical quantization, architectural extension, PASM-based mapping, multi-level verification, and architectural evaluation within a unified framework.

Chapter 3

Method

This chapter presents the methodology adopted for deploying the DANSE prior computation on the SiLago architecture. Rather than redesigning or retraining the DANSE model, the objective of this work is to transform a pretrained DANSE implementation into a hardware-executable representation suitable for execution on the Dynamically Reconfigurable Resource Array (DRRA). To achieve this objective, the computational structure of the neural network is analyzed and reformulated into a set of hardware-friendly matrix-vector multiplication kernels. The extracted kernels are subsequently converted to fixed-point representation and mapped onto the target architecture using PASM instructions. Finally, a multi-level verification flow is employed to ensure functional correctness across software, architectural simulation, and RTL implementation levels.

3.1 Data Collection and Dataset Generation

The evaluation in this thesis is based on synthetically generated datasets that simulate nonlinear dynamical systems. These datasets are constructed to provide controlled and reproducible input sequences for training and testing the DANSE model.

3.1.1 State-Space Model

The data generation process follows a discrete-time nonlinear state-space model defined as:

$$x(t+1) = F(x(t)) + e(t), \tag{3.1}$$
$$y(t) = H(x(t)) + w(t), \tag{3.2}$$

where $x(t)$ represents the hidden state vector, $y(t)$ is the observation vector, $F(\cdot)$ is a nonlinear state transition function, and H is the observation matrix. The process noise $e(t)$ and measurement noise $w(t)$ are assumed to follow Gaussian distributions.

3.1.2 Dynamical Systems

To generate nonlinear state trajectories, chaotic attractor systems are used, including the Lorenz system, the Chen system, and the Lorenz-96 system.

These systems are chosen due to their nonlinear and time-varying dynamics, which provide challenging benchmarks for nonlinear state estimation algorithms. The Lorenz and Chen systems are three-dimensional systems, while the Lorenz-96 system extends to higher-dimensional dynamics.

The nonlinear transition function $F(\cdot)$ is implemented using a discretized approximation of the continuous-time system dynamics.

3.1.3 Dataset Generation

The datasets are generated using Python-based simulation tools. For each experiment, a set of trajectories is produced by simulating the state-space model over a fixed time horizon.

The dataset generation process is characterized by the following parameters:

- T: sequence length
- N: number of trajectories
- σ_e^2: process noise variance
- SMNR: signal-to-measurement-noise ratio

Each dataset consists of state sequences $X \in \mathbb{R}^{N\times(T+1)\times m}$ and observation sequences $Y \in \mathbb{R}^{N\times T\times n}$, where m and n denote the dimensions of the state and observation vectors, respectively.

3.1.4 Training and Testing Setup

The generated datasets are divided into training and testing subsets. The DANSE model is trained using observation sequences $y(1 : t)$, and the estimated state $\hat{x}(t)$ is compared against the ground-truth state $x(t)$.

Performance evaluation is conducted by comparing predicted states with true states across the test dataset using metrics such as mean squared error (MSE) and normalized mean squared error (NMSE).

3.2 Application Analysis and Partitioning

The objective of this stage is to transform the trained DANSE model into a hardware-oriented representation suitable for deployment on the SiLago architecture. Rather than directly mapping the original neural-network implementation, the computational structure of DANSE is first analyzed and decomposed into smaller computational units. Through this process, the dominant computational kernels are identified and reformulated into reusable arithmetic operations that can be efficiently executed on the DRRA architecture.

The methodology follows a top-down analysis approach. Starting from the trained DANSE model, the computational workload is progressively decomposed into prior and posterior computations, followed by extraction of dominant arithmetic kernels. The resulting kernel representation serves as the foundation for the subsequent quantization and hardware mapping stages.

3.2.1 Floating-Point Baseline

The starting point of the reformulation process is a trained DANSE model implemented using floating-point arithmetic. The model is evaluated using double-precision (FP64) representation, which serves as the numerical reference throughout the transformation process.

The floating-point baseline is used to establish the prediction accuracy of the original implementation and to provide a reference for evaluating the impact of quantization and hardware-oriented reformulation.

3.2.2 Parameter Extraction

After training, the parameters of the model are extracted for further analysis. These include the weights and biases of the GRU layers as well as the

parameters of the fully connected layers used to compute the mean and variance outputs.

The GRU parameters consist of input-to-hidden and hidden-to-hidden weight matrices organized into reset, update, and candidate gates. The fully connected layers consist of linear transformations applied to the hidden state.

All extracted parameters are stored and used for fixed-point quantization and hardware-oriented mapping.

3.2.3 Workload Analysis and Model Decomposition

To enable hardware-oriented implementation, the computational structure of the DANSE model is explicitly decomposed into fundamental arithmetic operations.

The DANSE model consists of two major computational stages: the prior update and the posterior update. The prior update is implemented using a GRU-based recurrent neural network followed by fully connected layers. This stage contains repeated matrix-vector multiplication operations and therefore represents the dominant computational workload of the model.

The posterior update refines the state estimate using Bayesian update equations. Under the assumptions used in this work, the observation matrix is chosen as an identity matrix, while both the prior covariance and measurement noise covariance are diagonal matrices. As a result, the posterior update reduces primarily to element-wise arithmetic operations.

This analysis shows that the computational complexity of the prior update is significantly higher than that of the posterior update. Therefore, the prior update becomes the primary target for hardware acceleration.

Based on this decomposition, the computation is reformulated into primitive operations including matrix-vector multiplication, vector addition, element-wise multiplication, and nonlinear activation functions. This representation enables direct mapping onto the DRRA execution model.

The decomposition further reveals that matrix-vector multiplication constitutes the dominant computational workload across both the GRU and fully connected layers. Consequently, matrix-vector multiplication becomes the primary target for hardware acceleration in this thesis. This observation directly motivates the kernel-extraction strategy described in the subsequent hardware-mapping stage.

3.2.4 Computational Kernel Extraction

Following the workload decomposition, the arithmetic workload of the DANSE model is analyzed to identify the dominant computational kernels.

The analysis reveals that the majority of computations appearing in both the GRU and fully connected layers can be represented as matrix-vector multiplication followed by bias addition:

$$y = Wx + b. \tag{3.3}$$

For the Lorenz system, the extracted kernels include:

- $[30, 3] \times [3, 1] + [30, 1]$
- $[30, 30] \times [30, 1] + [30, 1]$
- $[32, 30] \times [30, 1] + [32, 1]$
- $[3, 32] \times [32, 1] + [3, 1]$

For the Lorenz-96 system, the extracted kernels include:

- $[30, 20] \times [20, 1] + [30, 1]$
- $[30, 30] \times [30, 1] + [30, 1]$
- $[32, 30] \times [30, 1] + [32, 1]$
- $[20, 32] \times [32, 1] + [20, 1]$

Although these computations originate from different stages of the DANSE model, they share a common arithmetic structure based on matrix-vector multiplication and bias addition. Consequently, they are selected as the primary targets for hardware acceleration.

3.3 Numerical Representation Selection

The original DANSE model is implemented using double-precision floating-point arithmetic. While floating-point computation provides high numerical precision, it introduces significant hardware complexity when deployed on resource-constrained architectures.

To enable efficient hardware implementation on the DRRA architecture, a fixed-point representation is adopted in this work. Fixed-point arithmetic

can be efficiently realized using integer-based operations while maintaining sufficient numerical precision for state-estimation tasks.

The objective of this stage is therefore to determine an appropriate fixed-point representation that preserves the numerical behavior of the original DANSE model while remaining suitable for hardware deployment.

3.3.1 Quantization Procedure

The quantization process is applied to model parameters, input observations, intermediate variables, and arithmetic operations. Both the prior update and posterior update are reformulated using fixed-point arithmetic.

During this process, floating-point operations are replaced by fixed-point counterparts including integer-based addition, multiplication, scaling, and saturation operations. The implementation ensures that all computations are performed consistently under a unified fixed-point representation.

3.3.2 Fixed-Point Representation Design

The fixed-point representation follows the Qm.n format, where m and n denote the integer and fractional bit-widths, respectively.

The first step in the design process is range analysis. The numerical range of weights, biases, observation sequences, and intermediate variables is evaluated using the floating-point baseline. It is observed that most model parameters remain within a relatively small range, while the observation sequence may occasionally exceed the representable range due to measurement noise.

Instead of increasing the integer bit-width, saturation is applied to infrequent outliers during quantization. Experimental evaluation shows that the clipping of these outliers has limited impact on the final state estimation accuracy.

After selecting the integer bit-width, different fractional precisions are evaluated, including Q8.16 and Q8.8. Reduced-precision floating-point formats such as Float16 and Float8 are also evaluated for comparison.

The final representation is selected based on the trade-off between prediction accuracy and hardware implementation complexity.

3.3.3 Evaluation Method

The reformulated model is evaluated by comparing its output against the floating-point baseline. The evaluation focuses on the final state estimation

result rather than intermediate computations.

For each numerical representation, the estimated state $\hat{x}(t)$ is compared against the ground-truth state $x(t)$ using metrics such as mean squared error (MSE) and normalized mean squared error (NMSE).

The evaluation is performed under different noise conditions to analyze the robustness of the reformulated implementation.

3.4 Hardware Mapping Methodology

The objective of the hardware mapping process is to transform the computational kernels extracted from the DANSE model into reusable hardware kernels that can be executed on the SiLago architecture.

Rather than directly mapping the complete neural network, the mapping methodology follows a hierarchical approach. First, representative computational kernels are selected from the extracted workloads. Next, memory layouts and hardware resources are assigned according to DRRA constraints. Finally, the resulting implementation is verified through a multi-level validation flow.

This methodology enables the deployment process to remain independent of a specific DANSE configuration while maximizing resource reuse across different matrix dimensions.

3.4.1 Application Selection

The application partition analysis described in 3.2.4 reveals multiple matrix-vector multiplication kernels with different dimensions. Implementing a dedicated hardware accelerator for every matrix dimension would significantly increase design complexity and reduce hardware reuse.

To improve scalability, the extracted kernels are unified into a small number of generalized computation platforms.

The first platform adopts a generalized matrix-vector multiplication structure:

$$[32, 32] \times [32, 1] + [32, 1]. \tag{3.4}$$

This platform is capable of representing larger kernels appearing in the DANSE model, including recurrent-state updates and fully connected layers. Smaller dimensions can be accommodated through zero-padding without requiring architectural modifications.

However, kernels associated with low-dimensional observations exhibit significantly smaller input dimensions. Mapping these kernels directly onto the generalized platform would result in poor utilization because only a small subset of the available inputs would be used.

Therefore, a second platform is introduced:

$$[32, 3] \times [3, 1] + [32, 1]. \tag{3.5}$$

This platform is optimized for low-dimensional inputs while maintaining the same computational structure.

As a result, all matrix-vector multiplication kernels extracted from the DANSE model can be mapped onto one of two reusable hardware platforms. This significantly reduces implementation complexity while preserving compatibility with different model configurations.

3.4.2 Memory Layout for Input and Output Buffers

Since DRRA execution depends on explicit data movement, the input and output data must be arranged before resource binding and PASM scheduling are considered.

For each kernel, the input data are divided into three logical regions: weight matrix, input vector, and bias vector. The output data are represented as the resulting vector of the matrix-vector computation. This logical layout defines the order in which data are read from the input buffer and written to the output buffer.

The memory organization is designed according to the data width of the selected fixed-point representation and the transfer granularity of the DRRA buffer interface. Therefore, the mapping process must consider both arithmetic dimensions and memory packing. This step determines how the abstract operation

$$y = Wx + b \tag{3.6}$$

is represented as buffer regions that can later be consumed by the hardware implementation.

3.4.3 Model0 Reference Design

After the logical input and output memory layout is defined, a Model0 reference implementation is constructed for each unified kernel.

Model0 serves as the functional reference of the hardware mapping flow. It uses the same fixed-point representation and the same logical buffer organization as the hardware implementation. Therefore, Model0 verifies not only the arithmetic result, but also the interpretation of input and output buffer contents.

This step is important because it connects the software-level kernel definition with the hardware-oriented memory model. The Model0 implementation explicitly reflects how the input buffer is divided into weight, input-vector, and bias regions, and how the output buffer stores the computed result.

Model0 also provides a simplified view of the intended execution behavior. It defines the kernel functionality before low-level resource binding is introduced, including the row-wise dot-product structure and the independence between output elements. This makes it possible to reason about which parts of the computation can be reused, serialized, or executed in parallel during later DRRA mapping.

3.4.4 Resource Binding Strategy

After Model0 defines the functional and memory-level behavior of the kernel, the computation is mapped onto DRRA resources.

The binding process assigns different roles to available resources. Storage-related operations are mapped to register-file resources, arithmetic operations are mapped to data-processing resources, and communication operations are mapped to switch-box resources. The purpose of this step is to establish the correspondence between the logical kernel operations and the physical resources available in the DRRA fabric.

The binding strategy is guided by three rules. First, data that are reused during computation, such as input vectors and bias values, should be stored locally whenever possible. Second, arithmetic resources should be assigned according to the dominant operations of the kernel, especially multiplication, accumulation, and bias addition. Third, communication paths must be selected so that data dependencies between memory access, local storage, arithmetic execution, and output write-back are preserved.

This step does not yet define the complete PASM schedule. Instead, it determines the resource-level structure that the PASM implementation must realize. The detailed slot assignment, routing paths, and scheduling constraints are described in Chapter 5.

3.4.5 Arithmetic Adaptation

The selected fixed-point representation requires arithmetic behavior that is not directly provided by the original DRRA resource library. Therefore, the mapping methodology includes an arithmetic-adaptation stage.

In this work, a Q8.8 arithmetic component is introduced by extending the signed 16-bit integer execution model. The purpose of this component is to support fixed-point addition, multiplication, and multiply-accumulate operations required by the matrix-vector kernels.

The arithmetic adaptation is applied consistently at two implementation levels. At the SST level, the component provides an architectural simulation model used by Model2. At the RTL level, the corresponding hardware resource provides the behavior required by Model3 verification. This ensures that the same fixed-point arithmetic interpretation is used during both architectural simulation and RTL simulation.

The detailed implementation of the Q8.8 component, including instruction definition, operation modes, SST handlers, and RTL behavior, is presented in Chapter 5.

3.4.6 Verification Methodology

After memory organization, Model0 construction, resource binding, and arithmetic adaptation are defined, the kernel can be implemented using PASM.

The PASM implementation is responsible for realizing the mapping decisions made in the previous steps. It describes memory access, local storage, routing, arithmetic execution, and synchronization between operations. Timing dependencies are expressed through PASM constraints to ensure that data are available before they are consumed.

Correctness is evaluated through a three-level verification flow. Model0 represents the functional reference implementation. Model2 corresponds to SST-based architectural simulation, where the PASM program is executed on the DRRA architectural model. Model3 corresponds to RTL-level simulation, where the mapped design is verified using RTL resources.

The verification process checks whether the same fixed-point output is produced across the abstraction levels. Model0 validates the functional definition and buffer interpretation, Model2 validates the PASM mapping and architectural behavior, and Model3 validates the RTL implementation of the mapped hardware resources.

This staged verification methodology allows errors to be isolated according to abstraction level. For example, mismatches in Model0 indicate

functional or memory-layout errors, mismatches in Model2 indicate PASM or architectural-mapping errors, and mismatches in Model3 indicate RTL-level implementation inconsistencies.

Chapter 4

Software Implementation and Quantization

This chapter presents the implementation-oriented transformation flow used to prepare the DANSE model for DRRA-based hardware execution. The overall process consists of four major stages:

- Nonlinear dataset generation
- Computational decomposition of the DANSE forward pass
- Fixed-point quantization and numerical analysis

The software-level quantization process is not treated as an independent objective. Instead, it serves as a hardware-oriented reformulation stage that transforms the original floating-point DANSE model into a representation suitable for DRRA-based execution.

4.1 Pretrained Model and Evaluation Data

The objective of this thesis is not to develop a new state-estimation algorithm or training methodology, but to investigate the deployment of an existing DANSE implementation on the SiLago architecture. Therefore, the trained network parameters have been extracted from the trained model and are used throughout the deployment flow.

The DANSE model used in this work is trained offline using datasets generated from nonlinear dynamical systems. The resulting network parameters, including GRU weights, GRU biases, and fully connected layer

parameters, are extracted from the trained model and used throughout the software reformulation and hardware mapping process.

Similarly, the observation sequences used for evaluation are generated offline prior to hardware deployment. These datasets serve as input stimuli for quantization analysis, software verification, architectural simulation, and RTL simulation. Since the focus of this thesis is hardware-oriented deployment rather than model training, the details of dataset generation and network training are not considered further.

Consequently, the deployment flow begins with two pre-obtained inputs: a trained DANSE model and a corresponding evaluation dataset. Although the datasets are treated as pre-obtained inputs during the deployment process, the data generation framework remains fully configurable. As described in Chapter 3, parameters such as the sequence length T, the number of trajectories N, the process noise variance σ_e^2, and the signal-to-measurement noise ratio (SMNR) can be adjusted through the dataset generation scripts. This flexibility enables the generation of datasets with different system dimensions, noise conditions, and sequence characteristics for both model training and deployment evaluation. In this thesis, however, the generated datasets are used solely as input stimuli for the software reformulation and hardware verification flow.

4.2 Hardware-Oriented Decomposition of DANSE

The DANSE model contains both neural-network-based prior estimation and Bayesian posterior update stages. Before hardware mapping can be performed, the computational structure of the forward pass must be analyzed and decomposed into explicit arithmetic operations. This decomposition exposes the dominant computational kernels, identifies potential opportunities for hardware acceleration, and establishes the foundation for the subsequent fixed-point reformulation and DRRA-based implementation.

4.2.1 Forward-Pass Decomposition

To support hardware-oriented implementation, the original DANSE forward pass is manually decomposed into explicit computational steps. This decomposition exposes all parameters, intermediate variables, and dominant arithmetic operations involved in the forward computation.

The prior update is implemented using a GRU followed by fully connected layers. The GRU parameters are extracted from the trained model, including

the input-to-hidden weight matrix, hidden-to-hidden weight matrix, and their corresponding bias vectors. These matrices are divided into reset, update, and candidate gates. For each time step, the GRU computation is expressed as

$$\gamma_u = \sigma(W_{iu} y_{t-1} + b_{iu} + W_{hu} h_{t-1} + b_{hu}), \tag{4.1}$$
$$\gamma_r = \sigma(W_{ir} y_{t-1} + b_{ir} + W_{hr} h_{t-1} + b_{hr}), \tag{4.2}$$
$$\hat{h}_t = \tanh(W_{ic} y_{t-1} + b_{ic} + \gamma_r \odot (W_{hc} h_{t-1} + b_{hc})), \tag{4.3}$$
$$h_t = \gamma_u \odot h_{t-1} + (1 - \gamma_u) \odot \hat{h}_t. \tag{4.4}$$

After the GRU update, the hidden state is passed through one fully connected layer followed by two output layers:

$$f_t = \mathrm{ReLU}(W_f h_t + b_f), \tag{4.5}$$
$$m_{t|1:t-1} = W_m f_t + b_m, \tag{4.6}$$
$$v_{t|1:t-1} = \mathrm{softplus}(W_v f_t + b_v). \tag{4.7}$$

Here, $m_{t|1:t-1}$ represents the prior mean and $v_{t|1:t-1}$ represents the diagonal prior variance.

4.2.2 Posterior Simplification

The posterior update then incorporates the current observation y_t. With $L_{t|1:t-1}$ constructed from the predicted variance, the posterior computation is expressed as

$$R_t = H L_{t|t-1} H^T + C_w, \tag{4.8}$$
$$K_t = L_{t|t-1} H^T R_t^{-1}, \tag{4.9}$$
$$\epsilon_t = y_t - H m_{t|t-1}, \tag{4.10}$$
$$m_{t|t} = m_{t|t-1} + K_t \epsilon_t, \tag{4.11}$$
$$L_{t|t} = L_{t|t-1} - K_t R_t K_t^T. \tag{4.12}$$

The observation matrix H is chosen as an identity matrix, and the measurement noise covariance C_w is assumed to be diagonal. In addition, the prior covariance $L_{t|t-1}$ is represented as a diagonal matrix, where only the variance vector is predicted by the neural network.

Under these assumptions, all matrices involved in the posterior update remain diagonal. As a result, matrix multiplications and matrix inversion operations reduce to element-wise operations.

For example, the innovation covariance becomes:

$$R_t = \text{diag}(v_{t|t-1} + \sigma_w^2), \tag{4.13}$$

and its inverse is computed as an element-wise reciprocal. Similarly, the Kalman gain is given by:

$$K_t = \text{diag}\left(\frac{v_{t|t-1}}{v_{t|t-1} + \sigma_w^2}\right). \tag{4.14}$$

Consequently, all matrix operations in the posterior update reduce to element-wise arithmetic operations. This eliminates the need for matrix multiplication and matrix inversion, which are typically the most computationally expensive operations.

As a result, the posterior update has linear computational complexity with respect to the state dimension, while the prior update involves matrix-vector multiplications with higher computational cost. Therefore, the posterior update is not the dominant computational component of the DANSE model and does not require dedicated hardware acceleration. In contrast, the prior update represents the primary computational bottleneck and is the main target for optimization and hardware mapping.

4.3 Fixed-Point Quantization for Hardware Mapping

Following the computational decomposition, the floating-point implementation is transformed into a fixed-point representation suitable for hardware execution. The objective of this stage is to establish a numerical representation that can be efficiently supported by the arithmetic resources available in the DRRA architecture. This section introduces the fixed-point representation, arithmetic reformulation, range analysis, and precision exploration process used to evaluate candidate numerical formats for hardware deployment.

4.3.1 Q$m.n$ Representation

The fixed-point implementation uses a Q$m.n$ representation, where m denotes the number of integer bits and n denotes the number of fractional bits. A

floating-point value is first scaled by 2^n, rounded to the nearest integer, and then clipped to the representable range:

$$x_{\text{fixed}} = \text{clip}(\text{round}(x \cdot 2^n)). \tag{4.15}$$

The corresponding floating-point value can be recovered as

$$x_{\text{float}} = \frac{x_{\text{fixed}}}{2^n}. \tag{4.16}$$

4.3.2 Fixed-Point Arithmetic Reformulation

The software implementation defines arithmetic functions for fixed-point addition, multiplication, and matrix multiplication. Addition is performed directly on integer values with the same scaling factor. Multiplication requires rescaling after integer multiplication:

$$z_{\text{fixed}} = (x_{\text{fixed}} \cdot y_{\text{fixed}}) \gg n. \tag{4.17}$$

The same principle is applied to matrix multiplication.

Activation functions are evaluated by converting the fixed-point value back to floating-point, applying the activation function, and converting the result back to fixed-point. This approach is used for software-level quantization evaluation. Hardware acceleration of activation functions is not included in the current implementation.

4.3.3 Integer Range Analysis

Before selecting the fractional precision, the required integer bit-width is determined from range analysis. The range of model parameters, including GRU weights, GRU biases, and fully connected layer parameters, is recorded after model training. The range of the observation sequence Y is also analyzed, since Y is the direct input to the DANSE model.

The range analysis shows that most weights and biases are concentrated within a limited range, approximately within $[-4, 4]$. However, the observation sequence can have a significantly larger range, especially under noisy conditions or longer generated trajectories. Therefore, the integer bit-width must be selected to cover not only model parameters but also input observations and intermediate values.

4.3.4 Precision Exploration

After fixing the integer bit-width, different fractional bit-widths are tested to evaluate the trade-off between precision and implementation cost. In this work, fixed-point configurations with 16 fractional bits and 8 fractional bits are evaluated. These correspond to higher-precision and lower-precision fixed-point settings, respectively.

In addition to fixed-point formats, floating-point reduced-precision formats are also evaluated, including float16 and float8. These formats provide additional comparison points for understanding the accuracy loss caused by reduced numerical precision.

For each numerical format, both the prior update and posterior update are evaluated. The output of each quantized configuration is compared with the floating-point baseline to quantify the numerical error introduced by the representation.

Chapter 5

Hardware Mapping of DANSE on SiLago

This chapter presents the hardware implementation of the computational kernels extracted from the DANSE prior network. Following the software-level reformulation described in Chapter 4, the selected matrix-vector multiplication kernels are deployed onto the SiLago DRRA architecture.

The hardware implementation follows three main stages. First, the arithmetic resource library is extended to support fixed-point matrix-vector computation. In this stage, the original DPU resource is extended into a Q8.8-capable resource at both the SST simulation level and the RTL implementation level. The SST implementation provides an architectural simulation model, while the RTL implementation provides the corresponding hardware-level behavior.

Second, the extracted matrix-vector kernels are mapped onto the DRRA architecture using PASM. The PASM programs define the resource allocation, data movement, routing paths, repetition patterns, and synchronization constraints required to execute the kernels on the DRRA fabric. The kernel-level architecture configuration specifies the concrete cell structure and resource composition used during this mapping stage.

Finally, the implementation is verified through a multi-level simulation flow. Model0 is used as the software reference model, Model2 is used for SST-based architectural simulation, and Model3 is used for RTL simulation. The outputs from the SST and RTL simulations are compared against the software reference to validate that the fixed-point numerical behavior is preserved throughout the complete deployment flow.

The overall hardware implementation flow is summarized in Figure 5.1.

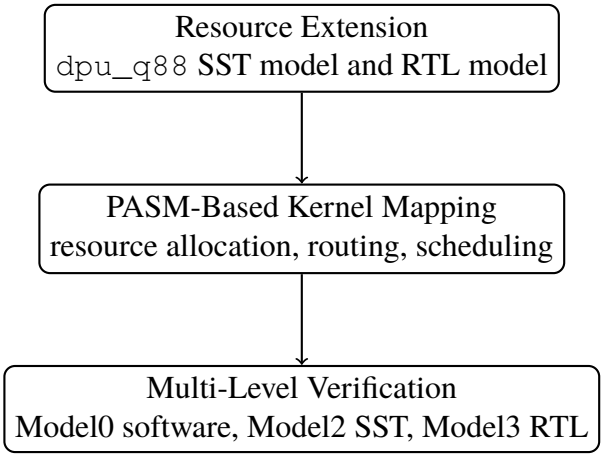

Figure 5.1: Overall hardware implementation flow.

5.1 DRRA Architecture and Resource Organization

The SiLago framework adopts a hierarchical architecture in which the DRRA fabric is constructed from reusable cells and resources. Each cell integrates communication, storage, and arithmetic resources that cooperate through the routing infrastructure of the array. This modular organization allows individual resources to be developed and extended independently while preserving compatibility with the overall architecture.

The implementation used in this work is organized under the `lib` directory, which contains the architectural building blocks of the framework. Major components include reusable resources, cell definitions, controllers, common utility modules, fabric-level integration logic, and verification infrastructure. Figure 5.2 illustrates the organization of the repository relevant to this work.

Among these components, the `resources` directory contains the reusable hardware resources that can be instantiated within a DRRA cell. Existing resources such as `swb` provide communication functionality, `rf` provides local storage, and `iosram` resources provide access to external memory structures. Arithmetic computation is performed by DPU-based resources.

To support fixed-point execution, the original DPU resource was extended to create a new arithmetic resource named `dpu_q88`. The newly introduced resource follows the same organizational structure as existing DRRA resources

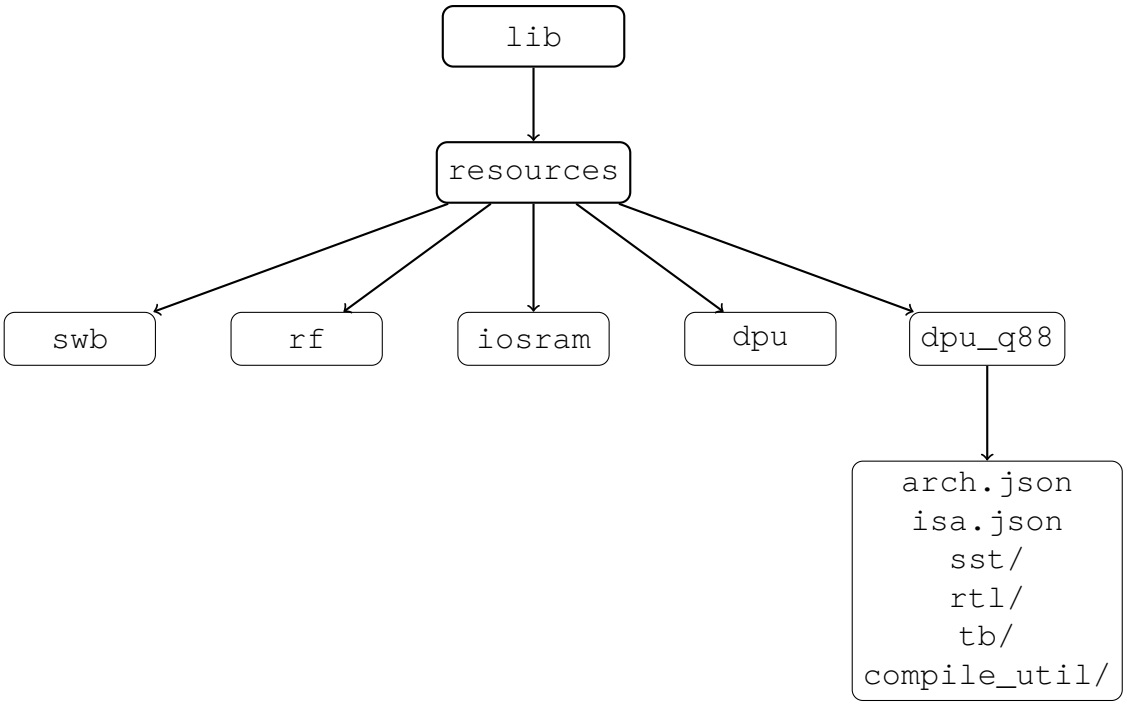

Figure 5.2: Repository organization of reusable DRRA resources and the proposed `dpu_q88` extension.

and contains the architectural metadata, instruction-set definitions, SST simulation model, RTL implementation, timing-model generation utilities, and resource-level verification infrastructure required by the SiLago toolchain.

This organization enables the resource to be integrated consistently into architectural simulation, RTL verification, and PASM-based deployment flows while maintaining compatibility with existing DRRA components.

5.2 Instruction Set Extension and PASM Operations

Within the SiLago framework, DRRA resources are controlled through PASM instructions. These instructions describe memory access, arithmetic execution, communication, and synchronization behavior, forming the interface between the mapped application and the underlying hardware resources.

To support fixed-point matrix-vector computation, the instruction set of the original DPU resource was extended to accommodate Q8.8 arithmetic operations. The extended instruction set preserves compatibility with the existing DRRA execution model while introducing the fixed-point addition, multiplication, and multiply-accumulate (MAC) operations required by the proposed accelerator.

In addition to arithmetic execution, PASM programs rely on several resource-level operations to coordinate data movement and execution across the DRRA fabric. Memory access is performed through `dsu` instructions, arithmetic operations are executed through `dpu` instructions, and communication is realized through `route` and `swb` instructions. Repetitive execution patterns are expressed using `rep`, while synchronization between resources is achieved through `evt` instructions.

Together, these operations provide the functionality required to implement matrix-vector multiplication kernels on the DRRA architecture. The detailed implementation of the extended DPU instruction set and the corresponding PASM mappings are presented in Table 5.1.

Table 5.1: Representative PASM operations used in this work.

Operation	Function
dsu	Memory access and data transfer
dpu	Arithmetic execution
route	Inter-cell communication
swb	Intra-cell communication and routing
rep	Loop repetition and iteration control
evt	Event-based synchronization

5.3 SST-Based Q8.8 Resource Implementation

After defining the DRRA architecture and instruction behavior, the next stage is to implement the arithmetic execution behavior of the configurable Q8.8 resource at the SST simulation level. The SST implementation provides the architectural simulation model of the `dpu_q88` resource used during Model2 verification. It extends the original DPU behavior with fixed-point arithmetic while preserving the standard DRRA resource interface for instruction decoding, event scheduling, FSM-controlled execution, and narrow-data communication.

5.3.1 Structure of the Q8.8 DPU Resource

The Q8.8 arithmetic implementation is mainly contained in the following files:

- `dpu_Q88.cpp`
- `dpu_Q88.h`
- `dpu_Q88_operations.cpp`
- `dpu_Q88_operations.h`

The `dpu_q88` component is implemented as a subclass of `DRRAResource`. It reuses the base DRRA resource infrastructure for ports, instruction handling, event scheduling, and data-buffer management. Internally, the resource maintains a set of FSM handlers, immediate buffers, operation handlers, and a dedicated accumulation register used by the multiply-accumulate operation.

Incoming narrow-data events are stored in local data buffers according to their slot identifiers. When the scheduled execution cycle is reached, the active

FSM handler invokes the selected DPU operation. This structure allows PASM instructions to configure arithmetic behavior while preserving the event-driven execution model of the DRRA framework.

5.3.2 Data Conversion and Fixed-Point Arithmetic

The SST model represents Q8.8 values using signed 16-bit integers. Since SST events transmit raw byte vectors, helper functions are used to convert between byte-level payloads and integer-level arithmetic operands.

The function `vectorToInt16()` converts input byte vectors into signed 16-bit values, while `int16ToVector()` converts arithmetic results back to byte-vector format. For the accumulation path, `vectorToInt32()` and `int32ToVector()` are used to represent the internal 32-bit accumulator.

The basic arithmetic operations are implemented using saturated fixed-point arithmetic. Addition and subtraction are performed directly on Q8.8 values, while multiplication first produces an extended intermediate product and then shifts the result right by eight fractional bits:

$$z_{\text{Q8.8}} = \text{sat}_{16}\left((x_{\text{Q8.8}} \cdot y_{\text{Q8.8}}) \gg 8\right). \tag{5.1}$$

Here, sat_{16} denotes signed 16-bit saturation. This ensures that the arithmetic output remains within the representable Q8.8 range.

5.3.3 Multiply-Accumulate Operation

The multiply-accumulate operation forms the core arithmetic primitive for matrix-vector multiplication. In the SST implementation, each MAC operation first multiplies two signed Q8.8 operands and rescales the product by shifting right by eight bits. The scaled product is then accumulated into a 32-bit accumulation register:

$$\text{ACC}_{next} = \text{sat}_{32}\left(\text{ACC} + (x \cdot y) \gg 8\right). \tag{5.2}$$

The updated accumulator value is stored internally in byte-vector format. The output of the MAC operation is obtained by saturating the updated accumulator back to a signed 16-bit Q8.8 value:

$$z = \text{sat}_{16}(\text{ACC}_{next}). \tag{5.3}$$

This design allows repeated MAC operations to preserve a wider internal accumulation range while still producing Q8.8-compatible outputs for

communication with other DRRA resources.

5.3.4 Operation Handler Structure

The SST implementation adopts a handler-based execution model. Each DPU mode defined in the ISA specification is associated with a dedicated operation handler. During execution, the instruction decoder extracts the operation mode from the incoming instruction and installs the corresponding handler into the active FSM configuration.

The implemented arithmetic handlers include:

- `IDLE`
- `ADD`
- `ADD_CONST`
- `SUBT`
- `SUBT_ABS`
- `MULT`
- `MULT_CONST`
- `MAC`
- `LD_IR`

This handler-based organization separates instruction decoding from arithmetic execution. It also ensures that the behavior of each PASM DPU instruction is mapped consistently to the corresponding SST operation.

5.3.5 Event Scheduling and Execution Behavior

In addition to arithmetic instructions, the SST resource also handles PASM control instructions such as `evt`, `rep`, `repx`, and `trans`. These instructions configure the timing behavior of DPU execution and reset events through the AGU-based scheduling mechanism.

The `evt` instruction defines events for DPU execution or accumulator reset. In the reset case, the internal accumulation register is cleared. The `rep` and `repx` instructions configure repeated execution patterns, while `trans`

specifies timing transitions. During simulation, these scheduling instructions determine when the configured arithmetic handler is activated.

The SST implementation therefore models not only the arithmetic behavior of the `dpu_q88` resource, but also its interaction with PASM-controlled timing and event scheduling. This makes the SST model suitable for Model2 architectural verification before the same behavior is reproduced at the RTL level in Model3.

5.4 RTL-Based Resource Implementation

The SST implementation described in the previous section provides an architectural-level representation of the proposed arithmetic resource. To enable hardware-level verification, the same functionality is implemented using synthesizable SystemVerilog templates within the SiLago RTL generation flow.

The RTL implementation preserves the arithmetic semantics defined by the SST model while introducing cycle-accurate behavior suitable for Model3 verification. The resource is constructed from a collection of reusable modules responsible for instruction decoding, timing control, mode selection, arithmetic execution, and accumulation management.

5.4.1 RTL Resource Structure

The top-level RTL implementation is centered around the `dpu.sv` module, which integrates instruction handling, timing control, and arithmetic execution into a single DRRA resource.

The incoming instruction stream is first decoded by the instruction decoder, which identifies DPU, EVT, REP, REPX, and TRANS instructions. The controller translates scheduling-related instructions into AGU configurations, while the mode selector stores the arithmetic configuration associated with each FSM state.

Two AGUs are instantiated within the resource. The first AGU controls arithmetic execution timing, while the second AGU is used to generate accumulator reset events. This organization mirrors the event-driven execution model implemented in the SST resource.

The resulting datapath separates instruction processing, scheduling control, and arithmetic execution, allowing PASM-controlled kernels to be executed using the same programming model adopted by the SST implementation.

5.4.2 Fixed-Point Arithmetic Modules

Dedicated arithmetic modules are used to implement Q8.8 addition and multiplication. Common numerical parameters, including the word width, accumulator width, fractional-bit count, and saturation limits, are defined in the package file and shared across all arithmetic modules.

The adder module performs signed fixed-point addition. Since both operands use identical Q8.8 scaling, the addition can be performed directly without rescaling. To prevent numerical overflow, the result is optionally saturated to the representable 16-bit range.

The multiplier module implements signed fixed-point multiplication. Multiplying two Q8.8 operands produces a Q16.16 intermediate result. Therefore, the product is arithmetically shifted right by eight bits before being returned to the Q8.8 domain:

$$z_{\text{Q8.8}} = \text{sat}_{16}\left((x_{\text{Q8.8}} \cdot y_{\text{Q8.8}}) \gg 8\right). \tag{5.4}$$

The use of signed arithmetic and arithmetic right shifting ensures that negative values are handled consistently with the SST implementation.

5.4.3 Multiply-Accumulate Datapath

The multiply-accumulate operation forms the core arithmetic primitive used by the matrix-vector multiplication kernels.

Unlike the standalone addition and multiplication operations, MAC execution maintains an internal 32-bit accumulation register. During each MAC cycle, the scaled product is added to the current accumulator value:

$$\text{ACC}_{next} = \text{ACC} + \left((x \cdot y) \gg 8\right). \tag{5.5}$$

The accumulator width is intentionally expanded from 16 bits to 32 bits to reduce overflow risk during long accumulation sequences. After accumulation, the result is converted back to a 16-bit Q8.8 value through saturation:

$$y = \text{sat}_{16}(\text{ACC}_{next}). \tag{5.6}$$

To support repeated dot-product evaluations, the accumulator can be cleared through a dedicated reset event generated by the second AGU. This mechanism allows PASM programs to explicitly define accumulation boundaries while preserving deterministic execution timing.

5.4.4 Consistency with the SST Model

The RTL implementation is designed to reproduce the arithmetic behavior of the SST resource model. Both implementations use signed 16-bit Q8.8 inputs and outputs, perform multiplication using an arithmetic right shift of eight bits, apply identical saturation rules, and maintain a 32-bit accumulation path for MAC operations.

Furthermore, the event-driven scheduling model implemented through EVT, REP, REPX, and TRANS instructions is preserved at the RTL level through the AGU-based timing infrastructure.

As a result, the same PASM kernel can be executed in both Model2 and Model3 without modification. Agreement between SST and RTL simulation results therefore provides evidence that the hardware implementation faithfully reproduces the behavior defined by the architectural model.

5.5 PASM-Based Mapping of Matrix-Vector Kernels

After extracting the dominant matrix-vector computations from the DANSE prior update, two reusable PASM mapping templates are implemented. The first template targets the generalized matrix-vector kernel

$$y = Wx + b, \tag{5.7}$$

with $W \in \mathbb{R}^{32\times32}$, $x \in \mathbb{R}^{32\times1}$, $b \in \mathbb{R}^{32\times1}$, and $y \in \mathbb{R}^{32\times1}$. The second template targets the smaller-input kernel with $W \in \mathbb{R}^{32\times3}$, $x \in \mathbb{R}^{3\times1}$, $b \in \mathbb{R}^{32\times1}$, and $y \in \mathbb{R}^{32\times1}$.

Although the matrix dimensions are different, both kernels share the same computation pattern: a row-wise dot product followed by bias addition. Therefore, they are mapped using the same cell-level structure, with different data-transfer and scheduling parameters.

5.5.1 Initial Three-Cell Mapping

The initial PASM implementation adopted a three-cell organization, as illustrated in Figure 5.3. The first cell was used as an input cell responsible for receiving data from the input buffer. The second cell performed the matrix-vector computation and bias addition. The third cell received the computed output and forwarded the result to the output buffer.

This organization provided a clear separation between communication and computation. Input staging, arithmetic execution, and output handling were distributed across dedicated cells, simplifying the early development and verification process. Inter-cell communication was realized through explicit PASM routing instructions connecting neighboring cells.

Although functionally correct, the three-cell organization introduced additional inter-cell communication and required dedicated RF resources for input and output staging.

5.5.2 Optimized Single-Cell Mapping

After validating the functionality of the original design, the mapping was further optimized by consolidating communication and computation into a single DRRA cell, as illustrated in Figure 5.4.

The optimized mapping reuses RF resources for both input staging and output storage, eliminating the need for dedicated input and output cells. As a result, all matrix-vector operations, bias addition, and result storage are performed within the same cell.

The single-cell organization significantly reduces inter-cell communication overhead because all arithmetic resources and RF resources are connected through intra-cell routing. The mapping employs dedicated RF resources for storing the weight matrix, input vector, and bias vector, while two DPU resources are allocated for multiply-accumulate operations and two DPU resources are allocated for bias addition and output generation.

Compared with the original three-cell organization, the optimized mapping reduces the number of utilized cells from three to one and eliminates inter-cell data transfers. This improves resource utilization and simplifies PASM routing while preserving the same computational behavior.

5.5.3 Input Buffer Layout

The input buffer stores the weight matrix, input vector, and bias vector in sequence. The layout is:

$$W \rightarrow x \rightarrow b. \tag{5.8}$$

For the 32×32 kernel, the weight matrix contains $32 \times 32 = 1024$ Q8.8 values. Since one bulk transfer contains 256 bits, corresponding to 16 Q8.8 values, the weight matrix occupies 64 bulk blocks. Therefore, the input vector and bias vector start at address 64.

For the 32×3 kernel, the weight matrix contains $32 \times 3 = 96$ Q8.8 values. This requires 6 bulk blocks. Therefore, the input vector and bias vector start at address 6.

This difference is reflected in the PASM initialization address. The 32×32 kernel uses `init_addr=64` for reading x and b, while the 32×3 kernel uses `init_addr=6`.

Table 5.2: Input buffer layout for the two PASM kernels

Kernel	**W values**	**W bulk blocks**	**Start address of x, b**
32×32	1024	64	64
32×3	96	6	6

5.5.4 Bulk Data Transfer

Bulk data transfer is used to move vectors and matrix data between cells and buffers. In both PASM implementations, the bulk communication path follows the same structure.

The input buffer first transfers data to `slot1`. The data are then forwarded through `slot2` to the computation cell. In the computation cell, the same bulk input path is reused to load different data types into different RF resources.

For the 32×32 kernel, the input vector x requires two bulk transfers, and the bias vector b also requires two bulk transfers. For the 32×3 kernel, the input vector only requires one bulk transfer, while the bias vector still requires two bulk transfers because the output dimension remains 32.

The output results are first stored in `slot2`. They are then transferred to `slot1` and finally written to the output buffer through the output cell.

5.5.5 Narrow Dataflow inside the Computation Cell

Narrow data communication is used to transfer individual Q8.8 values between RFs and DPUs. The narrow-data paths are configured by the switch box.

The input vector value is sent from `slot7` to the MAC DPU, while the corresponding weight value is sent from `slot6` to the MAC DPU. The MAC DPU computes and accumulates products of the form:

$$s_i = \sum_j W_{ij} x_j. \tag{5.9}$$

After the dot product is completed, the accumulated result is sent to the ADD DPU. At the same time, the corresponding bias value is read from `slot5` and sent to the ADD DPU. The final output is computed as:

$$y_i = s_i + b_i. \tag{5.10}$$

The result is written back to `slot5`, which is reused as the output RF.

5.5.6 Scheduling Difference Between the Two Kernels

The two PASM implementations mainly differ in the number of repeated operations required for the dot product. In PASM, `rep(iter=n)` generates $n + 1$ repetitions.

For the 32×32 kernel, each output element requires 32 MAC operations. Therefore, the inner computation loop uses 32 repetitions. The process is repeated for all 32 output rows.

For the 32×3 kernel, each output element requires only 3 MAC operations. Therefore, the input vector read and MAC scheduling are adjusted to match the three input dimensions. The output dimension remains 32, so the bias addition and output write-back are still repeated for 32 output elements.

The difference is summarized in Table 5.3.

Table 5.3: Scheduling difference between the two PASM kernels

Kernel	Input dimension	Output dimension	MACs per output
32×32	32	32	32
32×3	3	32	3

5.5.7 Routing and Constraint-Based Scheduling

PASM routing instructions are used to configure data movement between DRRA resources. A route instruction with `sr=0` defines a sending path, where the `source` field specifies the local slot and the `target` field specifies the destination through a routing bitmask. A route instruction with `sr=1` defines a receiving path, where the `source` field specifies the incoming route and the `target` field specifies the receiving slot.

In the original three-cell implementation, routing was primarily used for inter-cell communication. In the final single-cell implementation, the same routing mechanism is used mainly for intra-cell communication between RF resources and DPU resources. Consequently, routing remains an essential

component of the PASM program even though explicit inter-cell transfers are largely eliminated.

Timing is controlled using PASM constraints. The `cstr` statements specify ordering and synchronization relations between operations, such as ensuring that RF writes complete before arithmetic execution begins. Symbolic delay variables such as `t1`, `t2`, and `t3` are resolved by the constraint system during scheduling.

This constraint-based scheduling mechanism coordinates bulk-data transfers, RF loading, MAC execution, bias addition, and result write-back while maintaining deterministic execution behavior.

5.5.8 Summary

The two PASM kernels share the same computational structure and differ mainly in the number of multiply-accumulate operations required for each output element. The (32×32) kernel serves as a generalized platform for larger matrix-vector computations, while the (32×3) kernel provides a specialized implementation for low-dimensional input vectors.

Starting from an initial three-cell organization, the final implementation adopts a resource-reuse-oriented single-cell mapping strategy. By consolidating communication and computation within a single DRRA cell, the mapping reduces inter-cell communication, improves resource utilization, and simplifies routing while preserving the same computational behavior.

Together, these PASM implementations support the dominant matrix-vector operations appearing in the DANSE prior update and provide the basis for the SST and RTL verification results presented in the next chapter.

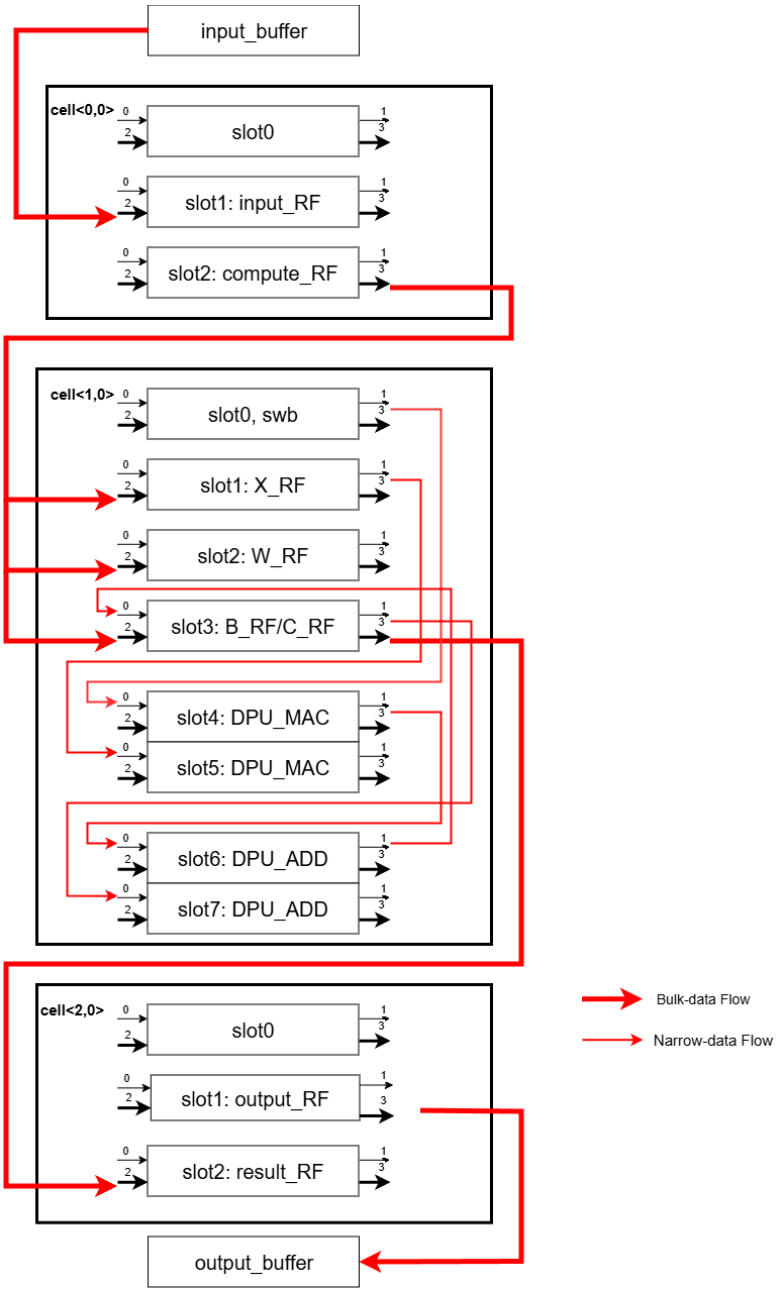

Figure 5.3: Initial three-cell mapping of the matrix-vector kernel.

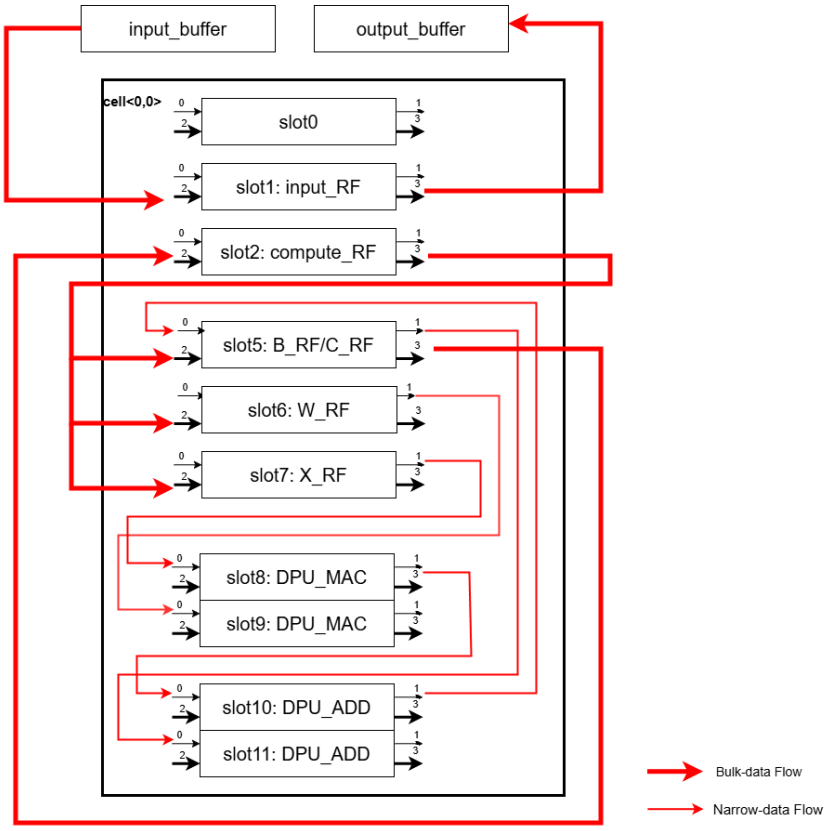

Figure 5.4: Optimized single-cell mapping used in the final implementation.

Chapter 6

Results and Analysis

6.1 Software Quantization Results

This section presents the software-level evaluation of numerical quantization for the DANSE model. The evaluation focuses on two aspects: first, determining a suitable integer bit-width based on the numerical range of model parameters and input observations; second, comparing the final state estimation accuracy under different numerical formats.

6.1.1 Integer Bit-Width Evaluation

Before evaluating different fractional precisions, the integer bit-width must first be determined to ensure that the quantized representation can cover the dynamic range of both model parameters and input observations.

The range analysis shows that most weights and biases of the trained DANSE model are concentrated within approximately $[-4, 4]$. Therefore, the model parameters themselves require only a small integer range. The main challenge instead comes from the observation sequence Y, whose range increases under noisy conditions and longer trajectories.

Although several observation samples exceed the range $[-128, 127]$, further analysis shows that these outliers occur infrequently and are mainly introduced by measurement noise. Most observation values remain within the 8-bit signed integer range. Therefore, instead of increasing the integer bit-width, an additional experiment is performed using an 8-bit integer range together with saturation.

In the quantization process, values exceeding the representable range are

saturated to the maximum or minimum representable value:

$$x_{\text{sat}} = \begin{cases} x_{\max}, & x > x_{\max}, \\ x_{\min}, & x < x_{\min}, \\ x, & \text{otherwise.} \end{cases} \tag{6.1}$$

Experimental results show that the saturation of these infrequent outliers has only a limited impact on the final state estimation accuracy. This result indicates that the DANSE model is relatively robust to occasional clipping introduced by quantization.

Based on this observation, an 8-bit integer range is considered sufficient for the fixed-point implementation. This choice reduces hardware cost while preserving acceptable prediction performance.

6.1.2 End-to-End State Estimation Results

After selecting the integer bit-width, different numerical formats are evaluated by comparing the final estimated state $\hat{x}(t)$ with the ground-truth state $x(t)$. The tested formats include Double64, Float16, fixed-point Q8.16, fixed-point Q8.8, and Float8. The Double64 result is used as the floating-point baseline.

Table 6.1: End-to-end state estimation performance under different numerical formats.

Format	SMNR (dB)	NMSE	NMSE Std	MSE (dB)
Double64	20	-20.1264	0.2691	4.5042
Float16	20	-20.1309	0.2693	4.4997
Q8.16	20	-20.1263	0.2691	4.5043
Q8.8	20	-20.2440	0.2758	4.3866
Float8	20	1.5535	0.0592	26.1841
Double64	-10	-5.3780	0.2021	19.2548
Float16	-10	-5.3772	0.2021	19.2557
Q8.16	-10	-5.3774	0.2014	19.2554
Q8.8	-10	-5.3837	0.2016	19.2491

The results show that Q8.16 achieves almost identical performance to the Double64 baseline. More importantly, Q8.8 also preserves the final state estimation accuracy under both SMNR = 20 dB and SMNR = -10 dB. The difference between Q8.8 and Double64 is small in the end-to-end prediction metrics, indicating that the DANSE model is robust to this level of fixed-point quantization.

In contrast, Float8 leads to a significant degradation under SMNR = 20 dB, showing that overly aggressive precision reduction is not suitable for reliable state estimation in this implementation.

6.1.3 Q-format Selection

Although Q8.16 provides higher numerical precision, it requires more fractional bits and therefore increases storage and arithmetic complexity. Q8.8 achieves comparable end-to-end state estimation accuracy while using fewer fractional bits. Considering both prediction accuracy and hardware implementation cost, Q8.8 is selected as the fixed-point format for the subsequent hardware-oriented implementation.

6.2 Functional Verification

After extending the DRRA architecture and implementing the fixed-point arithmetic kernels, functional correctness must be verified across different abstraction levels. The objective of this verification is to ensure that the architectural model and RTL implementation preserve the arithmetic behavior defined by the software reference model.

Unlike the DANSE evaluation presented in Section 6.1, the verification inputs used in this section are generated randomly within the numerical ranges observed for the input vector x, weight matrix W, and bias vector b. This approach increases test coverage by exercising a broader range of fixed-point values and arithmetic corner cases. Consequently, the focus of this section is functional consistency rather than end-to-end state estimation accuracy.

6.2.1 Verification Flow

The verification framework follows the hierarchical simulation methodology described previously. Model0 serves as the golden reference implementation, while Model2 and Model3 correspond to the SST-based architectural simulation and RTL implementation, respectively.

For each simulation level, the output memory image is written into a dedicated binary file. The generated outputs are then compared against the Model0 reference using bitwise comparison. A verification stage is considered successful only when the generated memory image is identical to the corresponding Model0 output.

Figure 6.1 illustrates the verification procedure adopted in this work.

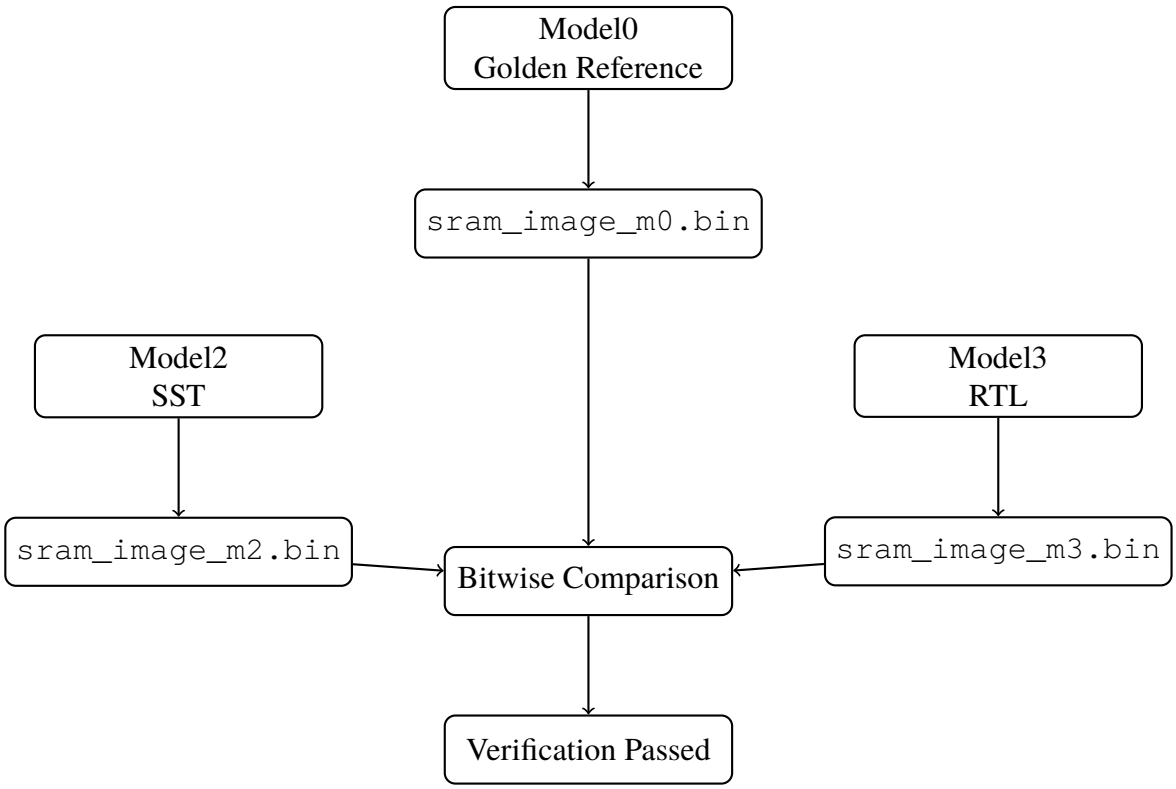

Figure 6.1: Verification methodology used for Model2 and Model3 validation.

This verification strategy ensures that both the SST model and the RTL implementation reproduce the exact fixed-point behavior defined by the software reference. Since both implementations are independently compared against Model0, successful verification demonstrates consistency across all abstraction levels.

6.2.2 Verification Results

Two matrix-vector multiplication kernels were evaluated using the verification framework:

- 32×3 matrix-vector multiplication with bias addition;
- 32×32 matrix-vector multiplication with bias addition.

For both kernels, the output memory images generated by Model2 and Model3 were compared against the corresponding Model0 reference outputs. No mismatches were detected during the bitwise comparison process.

Table 6.2 summarizes the verification results.

Table 6.2: Functional verification results.

Kernel	Model0 vs Model2	Model0 vs Model3	Result
32×3	Match	Match	Pass
32×32	Match	Match	Pass

Since all verification stages produced identical output memory images, only the final RTL outputs are presented here as representative examples.

For the 32×3 kernel, the RTL output buffers are

```
OB[0] = {2201, 64082, 64861, 64984, 2038, 1188, 2753, 459,
         62116, 725, 742, 1711, 65434, 2802, 2638, 63775}

OB[1] = {4225, 63547, 62596, 1110, 62771, 65505, 64937, 1345,
         560, 65233, 64730, 837, 64672, 2170, 64550, 64979}
```

For the 32×32 kernel, the RTL output buffers are

```
OB[0] = {63248, 902, 9042, 5616, 9822, 12951, 14377, 60017,
         62435, 63456, 58688, 62919, 247, 59922, 63329, 52393}

OB[1] = {6327, 3099, 64825, 51749, 243, 62599, 54936, 64311,
         7820, 56905, 2089, 13487, 4698, 1020, 53862, 58316}
```

The successful verification of both kernels confirms that the proposed arithmetic extensions, PASM mappings, SST models, and RTL implementations exhibit consistent fixed-point behavior across the entire verification hierarchy.

6.3 Architectural Evaluation

After functional correctness has been verified across Model0, Model2, and Model3, the implemented kernels are evaluated from an architectural perspective. The objective of this evaluation is not to benchmark absolute computational throughput, but rather to investigate how different DRRA mapping strategies affect resource utilization, communication overhead, and execution latency.

The evaluation focuses on two alternative mapping approaches developed during this work. The first approach employs a dedicated three-cell architecture, where input handling, computation, and output operations are distributed across separate cells. The second approach adopts a resource-reuse-oriented one-cell architecture, where all operations are executed within a single DRRA cell.

6.3.1 Execution Cycle Analysis

Table 6.3 summarizes the execution cycles measured for both matrix-vector multiplication kernels under the one-cell and three-cell mapping strategies.

Table 6.3: Execution cycle comparison between one-cell and three-cell mappings.

Kernel	One-cell	Three-cell
32×3	176	143
32×32	1100	1073

To establish a performance baseline, it is useful to first consider the theoretical computational complexity of the two kernels. For a matrix-vector multiplication, each output element requires a sequence of multiply-accumulate (MAC) operations. Consequently, the 32×3 kernel requires a total of $32 \times 3 = 96$ MAC operations, while the 32×32 kernel requires $32 \times 32 = 1024$ MAC operations. These values represent the theoretical lower bound of arithmetic execution.

The measured execution cycles are higher than the theoretical values because practical execution on the DRRA architecture involves more than arithmetic computation alone. Additional cycles are required for input data loading, scheduling overhead, inter-resource communication, and output data writing. Therefore, the observed execution latency consists of both computation time and architectural overhead.

For both kernels, the three-cell implementation achieves lower execution latency than the corresponding one-cell implementation. The improvement is particularly visible for the 32×3 kernel, where the execution time is reduced from 176 cycles to 143 cycles. A similar trend can also be observed for the larger 32×32 kernel, where the execution time is reduced from 1100 cycles to 1073 cycles.

To understand the source of this difference, the execution traces of the 32×32 kernel were further analyzed. It was observed that the first bulk-data transfer in the one-cell implementation begins at approximately cycle 67, whereas the three-cell implementation starts loading bulk data at approximately cycle 40. This difference aligns closely with the overall cycle gap between the two implementations (1100 vs. 1073 cycles), indicating that the performance difference can mainly be attributed to the initial scheduling and initialization phase rather than the steady-state computation phase.

This difference originates primarily from the initialization and scheduling phase preceding the computation. In the one-cell implementation, input loading, arithmetic execution, and output handling share the same local resources. As a result, additional scheduling effort is required to establish internal data paths and avoid conflicts between communication and computation activities before bulk-data streaming can begin.

In contrast, the three-cell implementation separates input, computation, and output functions across dedicated cells. This organization reduces resource contention during the initialization phase and allows data streaming to start earlier. Consequently, the computation pipeline becomes active sooner, which explains most of the observed cycle reduction.

6.3.2 Resource Utilization Analysis

Although the three-cell implementation achieves lower execution latency, this improvement comes at the cost of increased hardware resource consumption.

Table 6.4 summarizes the architectural characteristics of the two mapping strategies.

The most significant difference is the number of occupied DRRA cells.

Table 6.4: Comparison of one-cell and three-cell mapping strategies.

Feature	One-cell	Three-cell
DRRA cells	1	3
SWB resources	1	3
RF resources	3	3
DPU resources	2	2
Inter-cell routing	No	Yes
Resource reuse	High	Low
Communication overhead	Lower	Higher
Implementation complexity	Lower	Higher

The one-cell mapping executes the entire matrix-vector multiplication kernel within a single cell through extensive reuse of local RF, SWB, and DPU resources. In contrast, the three-cell implementation dedicates separate cells to input handling, arithmetic computation, and output generation.

As a consequence, the three-cell implementation requires three times more DRRA cells and additional inter-cell communication channels. Although this organization improves execution latency, it also increases routing complexity and overall hardware occupancy.

Considering that the execution-cycle improvement remains relatively modest for the evaluated kernels, the one-cell implementation provides a favorable tradeoff between performance and resource consumption. Therefore, the one-cell mapping was selected as the final implementation strategy in this work.

6.3.3 Comparison with Software Execution

The original DANSE implementation is developed and evaluated in Python using floating-point arithmetic. Although the numerical computations performed by the software and hardware implementations are equivalent, their execution models are fundamentally different.

In the software implementation, matrix-vector multiplication is executed as a sequence of arithmetic operations controlled by a processor. Resource allocation, scheduling, and communication are handled implicitly by the software environment and are largely hidden from the user.

The DRRA implementation exposes these architectural details explicitly. Storage resources, communication channels, arithmetic units, and routing operations must all be specified through PASM instructions. As a result, architectural decisions become visible and controllable during the implementation process.

Table 6.5 summarizes the conceptual differences between software execution and DRRA-based execution.

Table 6.5: Comparison between software execution and DRRA-based execution.

Aspect	Software Implementation	DRRA Implementation
Execution model	Sequential	Spatially distributed
Scheduling	Implicit	Explicit PASM scheduling
Communication	Hidden	Explicit routing operations
Resource allocation	Automatic	User-controlled
Hardware visibility	Limited	Fully exposed
Verification hierarchy	Software only	Model0–Model3
Hardware deployment	Not directly deployable	Deployment-oriented

While software execution provides a convenient environment for algorithm development and numerical evaluation, it does not provide visibility into architectural resource utilization or communication costs. In contrast, the DRRA implementation enables architectural exploration and allows different resource allocation strategies to be evaluated directly.

6.3.4 Discussion

The architectural evaluation demonstrates that multiple valid mapping strategies exist for the same computational kernel. The choice of mapping depends not only on execution latency but also on resource availability and implementation complexity.

The three-cell implementation achieves slightly lower latency through spatial separation of communication and computation. However, this benefit requires additional cells and more complex routing infrastructure. The one-cell implementation provides comparable performance while significantly reducing hardware resource requirements through aggressive resource reuse.

These observations suggest that resource-reuse-oriented mappings may be preferable when architectural resources are limited, whereas multi-cell mappings become increasingly attractive as the computational workload grows and communication overhead can be amortized over a larger number of arithmetic operations.

Overall, the proposed DRRA-oriented implementation methodology enables systematic exploration of these tradeoffs while preserving functional equivalence with the original software model.

Chapter 7

Conclusions and Future Work

This chapter summarizes the main findings of the thesis, discusses the limitations of the proposed approach, and outlines directions for future research. The conclusions are drawn with respect to the objectives and research questions defined in Chapter 1.

7.1 Conclusions

This thesis developed and evaluated a deployment methodology for quantized neural-network computations on the SiLago DRRA architecture. To achieve this objective, the work investigated numerical quantization, architectural extension, PASM-based mapping, multi-level verification, and architectural evaluation using representative matrix-vector multiplication kernels derived from the DANSE framework.

The first objective was to identify a hardware-friendly numerical representation capable of preserving the computational behavior of the target workload. Experimental evaluation showed that fixed-point representations can achieve prediction accuracy comparable to floating-point execution while significantly reducing arithmetic complexity. Among the evaluated formats, the selected fixed-point representation provided a suitable balance between numerical accuracy and implementation cost.

The second objective was to extend the DRRA architecture to support fixed-point neural-network computations. This was achieved through the development of a dedicated arithmetic resource and corresponding instruction-set extensions. The resulting implementation enables fixed-point multiplication, accumulation, and bias-addition operations to be executed efficiently within the SiLago framework.

The third objective was to investigate mapping strategies for representative neural-network kernels. Two alternative mapping approaches were developed and evaluated. The results demonstrated that different resource-allocation strategies lead to different tradeoffs between execution latency and hardware utilization. While the three-cell implementation achieved slightly lower execution latency, the one-cell implementation provided comparable performance with substantially reduced resource requirements.

The fourth objective was to establish a verification methodology across multiple abstraction levels. A verification flow spanning Model0, Model2, and Model3 was successfully implemented. Bitwise agreement was observed between the software reference model, the SST architectural simulation, and the RTL implementation for all evaluated kernels, demonstrating functional consistency across the complete implementation hierarchy.

Finally, the architectural evaluation demonstrated that deployment decisions cannot be based solely on execution latency. Resource utilization, communication overhead, and implementation complexity must also be considered. The results highlight the importance of architecture-aware mapping strategies when deploying neural-network workloads onto coarse-grained reconfigurable architectures.

Overall, the thesis demonstrates a complete workflow for transforming floating-point neural-network computations into hardware-oriented implementations suitable for deployment on the SiLago DRRA architecture.

7.2 Limitations

The results presented in this thesis demonstrate the feasibility of the proposed deployment methodology. However, the current work is subject to several limitations that constrain the scope of the conclusions that can be drawn. These limitations are discussed below in terms of implementation coverage, evaluation metrics, scalability, and architectural exploration.

7.2.1 Hardware Acceleration Scope

This thesis focuses on the deployment of matrix-vector multiplication and bias-addition kernels extracted from the DANSE framework, as these operations constitute the dominant computational workload of the prior network. The complete DANSE inference pipeline was not implemented on hardware. Components such as recurrent-network control logic, activation-function acceleration, and end-to-end runtime integration remain outside the

current implementation scope. As a result, the presented hardware design demonstrates the feasibility of accelerating representative DANSE kernels rather than the complete DANSE model.

7.2.2 Hardware Evaluation Metrics

The evaluation presented in this work primarily focuses on functional correctness and architectural behavior across software, SST, and RTL abstraction levels. While this verification flow is sufficient for validating the proposed arithmetic extension and mapping methodology, it does not provide a complete assessment of implementation cost. Metrics such as silicon area, power consumption, energy efficiency, and post-layout timing performance were not investigated and therefore remain open for future evaluation.

7.2.3 Scalability Evaluation

The experimental study covers a limited set of matrix-vector kernels derived from the target DANSE workload. These kernels were selected because they represent the dominant computation patterns observed in the model. However, the scalability of the proposed approach has not been systematically evaluated across a broader range of matrix dimensions and workload configurations. Additional experiments would be required to fully characterize the behavior of the architecture under different application scenarios.

7.2.4 Architectural Exploration

The architectural study investigates only a small set of mapping and resource-allocation strategies. Although the selected implementations demonstrate the feasibility of the proposed workflow, they do not represent an exhaustive exploration of the design space. Alternative communication schedules, routing strategies, and resource-binding schemes may lead to different performance-resource tradeoffs and therefore warrant further investigation.

7.3 Future Work

Several directions for future research can be identified based on the findings of this thesis.

7.3.1 Deployment of Complete Neural-Network Workloads

This thesis focuses on representative matrix-vector multiplication kernels extracted from the DANSE framework. A natural next step is the deployment of complete neural-network inference pipelines, including recurrent-network control logic, activation functions, and memory-management mechanisms. Such an implementation would enable evaluation of the full workload rather than individual computational kernels.

7.3.2 Automated Mapping and Code Generation

The PASM mappings developed in this work were created manually. As the complexity of neural-network workloads increases, manual mapping becomes increasingly difficult and error-prone. Future work could investigate automated compilation flows capable of translating quantized neural-network descriptions directly into PASM programs.

7.3.3 Extended Architectural Exploration

Only one-cell and three-cell mapping strategies were evaluated in this thesis. Future studies could explore additional resource-allocation strategies, larger DRRA fabrics, and alternative communication topologies. Such investigations could provide deeper insights into the relationship between architectural organization and workload characteristics.

7.3.4 Physical Implementation Evaluation

The current evaluation focuses on architectural functionality and execution behavior. Future work should investigate implementation metrics such as silicon area, power consumption, energy efficiency, and timing closure. These metrics are necessary for assessing the practical viability of the proposed approach in real hardware deployments.

7.3.5 Support for Additional Neural-Network Operations

The proposed arithmetic extensions primarily target matrix-vector multiplication and bias-addition operations. Future extensions could incorporate additional neural-network operators, such as activation functions, normalization

layers, and more complex recurrent-network primitives, thereby expanding the range of workloads that can be efficiently deployed on the DRRA architecture.

7.4 Reflections

From a technical perspective, this thesis contributes to improving the accessibility of hardware-oriented deployment methodologies for machine-learning workloads. By combining numerical quantization, architectural extension, mapping, and verification within a unified workflow, the work helps bridge the gap between algorithm development and hardware implementation.

The increasing deployment of machine-learning models in embedded and resource-constrained systems highlights the importance of efficient computing platforms. Architectures such as SiLago offer opportunities to improve computational efficiency while reducing hardware resource requirements. Such improvements may contribute to lower energy consumption and more sustainable computing infrastructures.

An important lesson learned during this project is that successful hardware deployment requires consideration of the entire implementation flow rather than isolated optimization of individual components. Numerical representation, architectural design, communication patterns, and verification methodology are strongly interconnected, and decisions made at one level often influence results at another. This insight may be valuable for future researchers working on hardware acceleration of machine-learning workloads.

References

[1] D. Pudi, S. Malviya, S. Boppu, Y. Yang, A. Hemani, and L. R. Cenkeramaddi, "Integer linear programming-based simultaneous scheduling and binding for silago framework," *IEEE Access*, vol. 12, pp. 124 081–124 094, 2024. doi: 10.1109/ACCESS.2024.3453503

[2] A. Ghosh, A. Honoré, and S. Chatterjee, "Danse: Data-driven non-linear state estimation of model-free process in unsupervised bayesian setup," in *2023 31st European Signal Processing Conference (EUSIPCO)*, 2023. doi: 10.23919/EUSIPCO58844.2023.10289946 pp. 870–874.

[3] S. Mittal, "A survey of fpga-based accelerators for convolutional neural networks," *Neural Computing and Applications*, vol. 32, no. 4, pp. 1109–1139, 2020. doi: 10.1007/s00521-018-3761-1

[4] X. Wang *et al.*, "Efficient hardware architectures for accelerating deep neural networks: Survey," *Electronics*, vol. 11, no. 24, p. 4102, 2022. doi: 10.3390/electronics11244102

[5] C. Zhang, P. Li, G. Sun *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *FPGA*, 2015. doi: 10.1145/2684746.2689060 pp. 161–170.

[6] S. Venieris and C. Bouganis, "Toolflows for mapping convolutional neural networks on fpgas," *ACM Computing Surveys*, vol. 51, no. 3, pp. 1–39, 2018. doi: 10.1145/3186332

[7] Y. Liu, "Hardware accelerator of matrix multiplication on fpga," Master's thesis, KTH Royal Institute of Technology, 2018.

[8] Y. Wang *et al.*, "Structured weight matrices-based hardware accelerators in deep neural networks," *IEEE Transactions on Emerging Topics in Computing*, 2018. doi: 10.1109/TETC.2018.2828960

[9] E. Nurvitadhi *et al.*, "A survey on hardware accelerators and optimization techniques for rnns," *Journal of Systems Architecture*, 2016.

[10] H. Kim *et al.*, "Stream processing dual-track cgra for object inference," *IEEE Transactions on VLSI Systems*, vol. 26, no. 11, pp. 2323–2336, 2018. doi: 10.1109/TVLSI.2018.2864117

[11] T. Guo *et al.*, "Ml-cgra: An integrated compilation framework for machine learning acceleration on cgras," in *FCCM*, 2022.

[12] L. Schönberger *et al.*, "Highly parameterised cgra architecture for design space exploration of machine learning applications," *Electronics*, vol. 13, no. 2, p. 401, 2024. doi: 10.3390/electronics13020401

TRITA-EECS-EX-2026:438
Stockholm, Sweden 2026

www.kth.se