



RICARDO MIGUEL OLIVEIRA MAFRA

Bachelor's in Micro and Nanotechnology Engineering

DELAY-LINE RESERVOIR COMPUTER SYSTEM-ON-CHIP

MASTER IN MICRO AND NANOTECHNOLOGIES ENGINEERING

NOVA University Lisbon
September, 2023

DELAY-LINE RESERVOIR COMPUTER SYSTEM-ON-CHIP

RICARDO MIGUEL OLIVEIRA MAFRA

Bachelor's in Micro and Nanotechnology Engineering

Adviser: Lars Keuninckx
Researcher, Imec

Co-advisers: Asal Kiazadeh
Doctorate Researcher, NOVA University Lisbon

Matthias Hartmann
Researcher, Imec

Delay-Line Reservoir Computer System-on-Chip

Copyright © Ricardo Miguel Oliveira Mafra, NOVA School of Science and Technology, NOVA University Lisbon.

The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

To the humans who assisted me in this journey, be it to let my thoughts out, be it a simple walk around the beautiful town of Leuven, or to introduce me to the power of coffee. To my dear friends back in Portugal, who have always been a reason to live and push forward, this work was also for you. And lastly, I dedicate this work to myself. To a lonely and self-hatred creature who could've fallen into the bottomless pit of despair and yet chose not to give in. Learning was a leeway to survival, but it had become much more; it had merged with oneself. With it, I learned self-love and became a better human. This work was a testament to that, a leap of courage into a subject I had close to no knowledge about, a test of my new self's wholeness. Accomplishing this work is proof that I am now capable of seeking the will to power within me.

ACKNOWLEDGEMENTS

First, thank you Lars for all the ideas you've contributed and for putting total trust in my competences. To work as freely as I did, while still having your support if needed. Matthias, thank you for the short free time windows you conceded to have a conversation about important life matters. You were a mentor during this internship, and, quite possibly one of the best at it. Lastly, Thanks to Asal for her assistance on administrative matters. Thank you three for everything.

I'd like to deeply thank a modern spartan who I was fortunate to meet during this phase of my life. Christos, an incredible human with whom I spent the very few breaks I had from work having the most thoughtful conversations. A friend who understood me like no other in such short-lived timeframe and made me see past my rigidity and self-criticism. I'm glad the universe dynamics allowed us to meet, but for now I'll wait to see you at Kilimanjaro's Piedmont.

To my new favourite living physicist, Nikos, our late-night kitchen conversations, together with Christos were proof of a level of affinity I hadn't seen happening in such little time. I adored every story and thought we shared, and thank you for trusting me. May the memories I possess endure the (perhaps) long journey ahead. I was ecstatic to finally meet someone who sees death, pain and evil in the exact same way I do. Let us have more deep conversations later in the timeline. We'll meet again in Kilimanjaro, the real Kilimanjaro.

Alex Romano, you have shown me the importance of having a break, even from my work. I burnt myself out for this project, more times than you have stolen the hammer of someone at Imec! But with your invitations to go for workouts and climbing I managed to recover and finish this dissertation. The summit of Kilimanjaro awaits us, until then thank you profoundly for everything.

Lastly, but with most importance, João. My very best friend, and possibly the only person in this universe that genuinely missed me. Thank you for not letting our friendship vanish, and for being there at all times that I wanted to let steam off. From my romances, to my unhealthy work ethic, thank you for going with me through all of it. I feel pity for all the Ricardo's in parallel universes that don't have you in their lives, you're marvellous.

” *“Brave and creative men never consider pleasure and pain as ultimate values—they are epiphenomena: one must desire both if one is to achieve anything.”*

— **Friedrich Nietzsche**, *The Will to Power*
(Philosopher)

ABSTRACT

As an increasingly digitized society materialises, it becomes a priority to process large volumes of data. Artificial intelligence particularly thrives the most when given access to large datasets of information, thus becoming extensively more potent these days. One such powerful technique is RNNs, typically applicable to forecasting and system modelling tasks. Traditionally these are resource-intensive models and thus often difficult to deploy as a System-on-Chip (SoC). In this work, we've taken a novel approach to RNNs — Delay Line Reservoir Computing (DLRC)— a resource-efficient machine learning paradigm to develop an FPGA-based machine learning accelerator. A first-approach implementation was done in software to serve as a target for the behaviour of the accelerator. The prototype performed up to standards under three distinct levels of complexity in time-series prediction and system identification tasks. A study on the impact of number format on performance was conducted and the 16-bit fixed-point format exhibited a good balance between resource usage and performance. The SoC architecture in our design comprises a MicroBlaze soft processor and a custom peripheral for the DLRC. In addition, an innovative and compact mask generator was designed to avoid the need to store the entire mask vector further improving resource efficiency. The behaviour of the hardware twin matched the intended behaviour in one-to-one simulations showing promising results towards an SoC implementation. With this work, we have proven the potential of using DLRCs to act as a digital twin of highly complex dynamic systems and as a powerful and resource-smart forecasting device.

Keywords: Machine Learning Accelerator, Delay-Line Reservoir Computing, System-On-Chip, FPGA, Neuromorphic

RESUMO

À medida que a sociedade se torna mais digital, o processamento de grandes volumes de dados torna-se uma prioridade. A inteligência artificial beneficia de maiores conjuntos de dados, tornando-se assim cada vez mais potente nos dias de hoje. Uma dessas poderosas técnicas são as RNNs, normalmente aplicáveis a tarefas de previsão e de modelação de sistemas. Tradicionalmente, são modelos que consomem muitos recursos e, portanto, são frequentemente difíceis de implementar em System-on-Chip (SoC). Neste trabalho, adoptámos uma nova abordagem às RNNs - Delay Line Reservoir Computing (DLRC) - um paradigma de aprendizagem automática eficiente para desenvolver um acelerador de aprendizagem automática baseado em FPGA. Foi feita uma implementação de primeira abordagem em software para servir de objetivo para o comportamento do acelerador. O protótipo teve um desempenho dentro dos padrões aceites em três tarefas distintas de previsão de séries temporais e de identificação de sistemas. Foi efectuado um estudo sobre o impacto do formato dos números no desempenho, e o formato de fixed-point de 16 bits apresentou o melhor equilíbrio entre a utilização de recursos e a perda de desempenho. Além disso, foi concebido um gerador de máscaras inovador e compacto para evitar armazenar todo o vetor de máscaras, melhorando ainda mais a eficiência dos recursos. A nossa implementação tem uma potência total no chip de 0,266 W, e a maioria dos recursos ficou abaixo de 5% do total disponível. O comportamento do hardware correspondeu ao pretendido em simulações um-para-um, mostrando resultados promissores para uma implementação SoC. Com este trabalho, provámos o potencial da utilização de DLRCs para atuar como um gémeo digital de sistemas dinâmicos altamente complexos e como um dispositivo de previsão poderoso.

Palavras-chave: Acelerador de Aprendizagem Automática, Delay-Line Reservoir Computing, FPGA, Computação Neuromórfica

CONTENTS

List of Figures	x
List of Tables	xiv
1 Introduction	1
1.1 Context	1
1.2 State-of-the-Art	2
1.3 Background	4
2 Materials and Methods	7
2.1 General Scope	7
2.2 Materials	7
3 Results	9
3.1 Experimental Setup	9
3.1.1 Python Prototype	9
3.1.2 Benchmarks	10
3.1.2.1 Overview	10
3.1.2.2 Lorentz Strange Attractor	10
3.1.2.3 Santa Fe Chaotic Time Series	11
3.1.2.4 Narma10 System Identification	11
3.1.3 Hardware prototype	12
3.1.3.1 System Overview	12
3.1.3.2 Block of Random Access Memory	13
3.1.3.3 System controller	14
3.1.3.4 Mask generator	15
3.1.3.5 Pipeline design	16
3.2 Discussion	18
3.2.1 Benchmarks	18
3.2.1.1 Lorentz Strange Attractor - Discussion	18

3.2.1.2	Santa Fe Time Series - Discussion	20
3.2.1.3	Narma10 - Discussion	20
3.2.2	Hardware prototype - Discussion	21
3.2.2.1	Mask Generator	21
3.2.2.2	Embedded application interface	23
3.2.2.3	Performance and Resource Utilization	24
4	Conclusions and Future Perspectives	26
	Bibliography	28
	Appendices	
A	Introductory Figures	31
B	Appendix 2 - Hardware Simulations	37
C	Appendix 3 - Benchmark Results	42
C.1	Lorentz Results	42
C.1.1	3D Plots	42
C.1.2	Time Series Plots	42
C.1.3	Regression Plots	42
C.1.4	9-SA Forecasting	42
C.1.5	Different initial condition	42
C.2	Santa Fe Results	42
C.3	Narma10 Results	42
D	Appendix 4 - Hardware Results	70
D.1	Mask Generator	70
D.2	Final	70
	Annexes	
I	Annex 1	77

LIST OF FIGURES

3.1	Bram Structural view. Note: Each row has 16 columns instead of 32 for simplicity of design	14
3.2	State Machine diagram for the implemented controller. Note: '-' denotes a "don't care".	15
3.3	Mask generator flow diagram. 5-LSB extraction from a number generated through the logistic map yields an address that is used to output the corresponding mask value.	16
A.1	SoC accelerator block design architecture. Vivado.	32
A.2	Delay Line Reservoir Computer IP block design architecture. Vivado.	33
A.3	Cobweb plot of an iterated one-dimensional map using a linear map. Notice how trajectories are dispersed towards infinity, never to return. This is the main reason for attractors to be nonlinear phenomenon.	34
A.4	Cobweb plot of an iterated one-dimensional map using the 3rd order polynomial in (state Eq.). Initial condition = 0.1999	35
A.5	Cobweb plot of an iterated one-dimensional map using the 3rd order polynomial in (state Eq.). Initial condition = 0.2. Notice how fast, "webs", or trajectories get separated despite starting close to each other. A.4	36
B.1	Structure in Embedded C to interface with the microcontroller in our SoC.	37
B.2	BRAM's response to the generation of output values. The signal's radix (SBM_DIA and src_current_dv) is Hexadecimal. The top 4 bytes are used for the delay value.	37
B.3	Retrieval of BRAM values, delay taps and weights. Read during Stage 3 of the Pipeline. Compare values at SBM_DOB with values computed at Stage 6	38
B.4	Timing diagram of the controller's response to inputs until production of addresses commences.	38

B.5	Timing diagram of the controller upon meeting ending criteria. The controller freezes once no more data remains in the buffer. The microcontroller ensures that there is always at least one input sample in the buffer until every input is processed.	38
B.6	Testbench code. Loading a six-valued mask to each of the 32 registers in the mask generator. Top are the values being written to the addresses below. . .	38
B.7	Masks being written as a result of the loading above	39
B.8	Initiation of the pipeline and iterative generation of mask values.	39
B.9	Timing diagram exhibiting the behaviour of the double buffer implemented at the start. Two inputs are inserted in the buffer before the start signal is raised. Once raised, the frontmost input enters the pipeline.	39
B.10	Timing diagram exhibiting the behaviour of the double buffer implemented during input transitions. As soon as the computation of the previous sample reach the end, the pipelined addresses reset, triggering st_vacant. The CPU then detects the vacancy, writes data to the hardware, and the buffer moves forward.	39
B.11	Timing diagram exhibiting the behaviour of the double buffer implemented at the end.	40
B.12	Pipelined registers in sync with the delay circuit. Reminder: Addresses in hardware have an offset of 4, as the two LSBs are ignored for 32-bit memories. For this reason the increment between addresses is equal to 4	40
B.13	Stage 2: Masking the input sample.	40
B.14	Stage 3: Reading weights and delay values from the BRAM, using ppl_addr2.	40
B.15	Stage 4: Combine masked samples with parent values as an input for the NL. For the first sample the BRAM remains empty, hence no parent value can be extracted.	41
B.16	Stage 4: For the second sample, parent values are now available. The transition between both samples is seen at the blue marker, when the input of the NL is no longer equal to the masked value.	41
B.17	Non-linearity. Computation in two steps for less overhead.	41
B.18	Value generation from Stage 6 of the Pipeline.	41
C.1	Comparison between ground truth and predicted attractor. 1-step ahead. Results from using FP numbers in a delay line with 256 VNs.	43
C.2	Comparison between ground truth and predicted attractor. 1-step ahead. Results from using FXP-Int16 numbers in a delay line with 256 VNs.	44
C.3	Comparison between the ground truth dynamics of each isolated variable and respective predictions. 1-step ahead. Results from using FP numbers in a delay line with 256 VNs.	45

C.4	Comparison between the ground truth dynamics of each isolated variable and respective predictions. 1-step ahead. Results from using FXP-Int16 numbers in a delay line with 256 VNs.	46
C.5	Linear Regression comparing target values with their respective prediction. Results from using FP numbers in a delay line with 256 VNs.	47
C.6	Linear Regression comparing target values with their respective prediction. Results from using FXP-Int16 numbers in a delay line with 256 VNs.	48
C.7	Effect of number precision in the performance of the DLRC. Horizontal lines correspond to best results from other experimental setups.	49
C.8	9-steps ahead. Comparison between ground truth and predicted attractor. Results from using FP numbers in a delay line with 256 VNs.	50
C.9	9-steps ahead. Comparison between the ground truth dynamics of each isolated variable and respective predictions. Results from using FP numbers in a delay line with 256 VNs.	51
C.10	9-steps ahead. Linear Regression comparing target values with their respective prediction. Results from using FP numbers in a delay line with 256 VNs.	52
C.11	Different Initial condition. Comparison between ground truth and predicted attractor. 1-step ahead. Results from using FP numbers in a delay line with 256 VNs.	53
C.12	Different Initial condition. Comparison between the ground truth dynamics of each isolated variable and respective predictions. 1-step ahead. Results from using FP numbers in a delay line with 256 VNs.	54
C.13	Linear Regression comparing target values with their respective prediction. Results from using FP numbers in a delay line with 256 VNs.	55
C.14	Comparison between ground truth and predicted time series. Results from using FP numbers in a delay line with 256 VNs.	56
C.15	Comparison between ground truth and predicted time series. Results from using FXP-Int16 numbers in a delay line with 256 VNs.	57
C.16	Linear Regression comparing target values with their respective prediction. 1-step ahead. Results from using FP numbers in a delay line with 256 VNs.	58
C.17	Linear Regression comparing target values with their respective prediction. 1-step ahead. Results from using FXP numbers in a delay line with 256 VNs.	59
C.18	Effects of number precision in the performance of the DLRC. Horizontal lines correspond to the best results from other experimental setups.	60
C.19	Effects of number precision in the performance of the DLRC. Horizontal lines correspond to the best results from other experimental setups.	61
C.20	Effects of number precision in the performance of the DLRC. Horizontal lines correspond to the best results from other experimental setups.	62
C.21	Comparison between the Narma10 and the mimicked systems obtained from the DLRC. Results from using FP numbers in a delay line with 256 VNs.	63

C.22	Comparison between the Narma10 and the mimicked systems obtained from the DLRC. Results from using FXP-Int16 numbers in a delay line with 256 VNs.	64
C.23	Regression Plot. Comparison between the Narma10 and the mimicked systems obtained from the DLRC. Results from using FP numbers in a delay line with 800 VNs.	65
C.24	Regression Plot. Comparison between the Narma10 and the mimicked systems obtained from the DLRC. Results from using FXP-Int16 numbers in a delay line with 800 VNs.	66
C.25	Time Series plot. Comparison between the Narma10 and the mimicked systems obtained from the DLRC. Results from using FP numbers in a delay line with 800 VNs.	67
C.26	Time Series plot. Comparison between the Narma10 system and the mimicked system obtained from the DLRC. Results from using FXP-Int16 numbers in a delay line with 800 VNs.	68
C.27	Effects of number precision in the performance of the DLRC. Horizontal lines correspond to the best results from other experimental setups.	69
D.1	Discrete distribution of generated mask addresses using a 5-LSB extraction. (From 8-bit FXP numbers) with the $x_0 = 0.3$	71
D.2	The “Offset Bias” effect. Discrete distribution of mask addresses obtained using 3-LSB extraction over 255 iterations of the logistic map. Three extracted bits represent up to 8 registers. If we consider a six-valued mask, then two of its values — say MV1 and MV3 — have to fill the two remaining registers. . . .	72
D.3	Distribution of generated mask-values using a 5-LSB extraction. Compared to D.1 we are closer to a uniform distribution as the discrepancy between MV1, MV3 and the remaining mask values is less severe.	73
D.4	Distribution of generated mask- addresses using a 3-LSB extraction.	74
D.5	Distribution of generated mask-addresses using a 5-LSB extraction.	75
D.6	Linear Regression comparing the output of the designed hardware with the target, the python model. The hardware retained a close match to the software, and for a multitude of samples.	76
I.1	Schematic of the Delay-Line Reservoir Computer. Solid lines highlight the discrete system’s first-order impulse response filter (IIR) Adapted from [11].	77
I.2	Comparison between RC architectures. Classical (top), Delay-Line (bottom).	78

LIST OF TABLES

3.1	One Step Ahead Forecasting for $X(t)$. Comparison with Multi-stage deep ESNs [19].	19
3.2	NIST: Set of statistical tests for discrete uniform distributions. Commonly used to validate Pseudo Random Bit Generators. On the rightmost column, there are 3 characters that correspond to the each of the three tests, F, CS and R respectively. "X"stands for failed the test for uniformity, while "Y"means that the test was successful. "Uniform" is the control. Tests performed at [21] . . .	23
3.3	Runtime comparison between software and hardware. Our design is at least 266 times faster than the fastest FP implementations.	24
3.4	Post-Implementation Resource estimates of the SoC single staged DLRC . . .	25

INTRODUCTION

1.1 Context

Humanity is moving towards an increasingly digitised society, predominantly caused by a large overflow of data in various industries. From patient-generated health data to vehicle-generated data and to highly detailed customer profiles, artificial intelligence, and particularly machine learning, thrives the most when given access to large datasets of information, thus becoming extensively more potent these days. Recurrent neural networks (RNNs) are a machine learning (ML) paradigm that excels at system modelling and time-series processing, such as weather and stock forecasting. However, their full potential remains to be explored, mainly due to implementation costs.

In context, RNNs can further enhance the benefits of hardware-in-the-loop (HIL) simulations. HIL is a technique that enables scalable and complete testing of highly complex embedded systems while avoiding the need for a fully assembled product. In this technique, manufactured components are replaced by an undistinguishable digital representation of these in hardware, a digital twin. This way, it is possible to expose the hardware to many real-life scenarios and fine-tune the system's controller without the time and cost implications of extensive physical tests. Consider a scenario where, rather than the arduous process of accurately representing the behaviour of an engine control unit in hardware, an RNN can learn its behaviour and readily become an ideal digital twin [25].

For these applications, there are a couple of well-studied neural network architectures, as is the case of Liquid State Machines (LSMs), Long Short-Term Memory networks (LSTMs) and Echo State Networks (ESNs), more recently collectively known as reservoir computing [9], [16]. This family of computation methods consists of neural networks with fixed interconnections between neurons and can be randomly attributed or connected in a regular Artificial Neural Network (ANN) fashion. Because trainable weights are only present in connections to the readout layer, a faster training process can be achieved by lowering the number of parameters to be optimised, lowering the complexity of the model

A reservoir computer can be described as a nonlinear high-dimensional dynamic

system that retains information from previous inputs, referred to as a transient response [9]. In practice, memory emerges by considering the values currently stored in neurons when processing new inputs, allowing the network to merge previous responses as new inputs arrive. One could picture these systems as a container full of water where tiny droplets are dropped. From each droplet, a perturbation can be observed on the water's surface. If dropped from multiple positions at different instances, these perturbations will interfere with each other until vanishing, some more prominent, others less. We, the observers, will see the transient response in this system at a given time, and that is what the readout nodes "perceive".

Typically, these neuromorphic algorithms are trained with generalised data to perform multiple distinct tasks, similar to our brains [16]. If the number of nodes in the reservoir is large, these systems map the input to a high-dimensional space. Therefore, the network must have a large number of neurons ($\approx 10^3$). For this exact reason, most implementations of these systems are done in software, as opposed to other machine learning techniques that have already been successfully embedded in specialised hardware units [2].

Naturally, reservoir computing characteristics impose some challenges to SoC implementations in the sense that it is significantly costly to implement in hardware. Not because of the implementation cost of each neuron but due to the sheer increase in area consumption from interconnections. Larger dies translate into a drop in production yield due to an increased probability of defect per die, thus leading to a higher cost-per-chip. Moreover, it is complex to describe its recurrent behaviour through parallel processing, which is essential to implement any system through a hardware description language such as Verilog or VHDL.

By migrating software-driven models to hardware, one can take advantage of the innate concurrency of the latter to enhance speed to a greater extent. Furthermore, since computation is performed at a lower level, it becomes much more efficient energy-wise, significantly lowering power consumption. Currently, transformers, such as a single large language model, have a carbon footprint well beyond what a single car consumes on average in its lifetime in the US [7]. These reasons make SoC implementations highly desirable for most machine-learning applications.

1.2 State-of-the-Art

LSTMs are RNNs with several control nodes and memory controllers to decide whether an input should be stored, forgotten, or used as output. In this case, more resources are required due to the addition of control gates required for long-term memory behaviour, therefore not addressing the problem. As a workaround, some architectural changes were proposed, leading to Economic-LSTMs.[12] These LSTMs rely on only one control gate to forget and update inputs, requiring fewer components and leading to 34% less area consumption reduction while outperforming regular LSTMs in latency and accuracy.

One immediate approach is to directly reduce the number of neurons in each network so that it effectively resolves the issue at hand. As one would expect, even though it allows for simpler and more cost-effective hardware implementations, it may need more accuracy for more complex tasks. An alternative strategy is to process each input in multi-parallel output layer nodes fed during different timesteps [14]. The architecture can be described as a single hidden layer and B output layers. Firstly, the hidden layer processes the inputs sequentially and feeds each output layer in parallel. This method takes advantage of the fact that the processing in the hidden layer has a much shorter execution time than the one in each output layer, leading to faster results than a two-stage pipeline architecture [13]. This architecture has also optimised data reuse and allocation such that a single language model can fit in an FPGA without consuming over 40% of any resource. A similar idea as the one described above is applied to LSTMs, the difference being that it uses a single module to receive simultaneously both the input and the last output, giving it the ability to measure errors [5].

Another possibility to reduce the complexity is to divide the total number of neurons per multiple smaller networks in parallel, whose output is then combined by a decoder into a single output. This method is also viable because complexity does not scale linearly with the number of neurons but rather exponentially. This way, one can avoid severe implications in area consumption while maintaining the same effective number of neurons [1]. However, this was only seen as a viable strategy for LSTMs since, for vanilla RNNs, improvements were only seen when the parameter size was increased to a few thousand in place of a few hundred.

In comparison, Delay-Line Reservoir Computing (DLRC) allows for a simpler physical implementation by creating virtual copies of a single neuron over a short time interval. If fast enough, time-multiplexing enables a scarce resource to be shared anew across time, saving essential resources in the process. Consequently, using high-speed components that would be utterly unfeasible otherwise due to implementation costs becomes a viable option [2]. For more details, see the Background subchapter.

When it comes to optimising the parameters of DLRCs, it was shown that in some instances, the choice of these can lead to a compromise where neither can be optimal simultaneously. For this exact matter, a strategy that was suggested is cascaded DLRCs. A cascaded approach allows for different tuning of the main parameters of each stage, functionalising them further. These bring many advantages, such as increasing the likelihood of all important hidden data features being caught and the ability to recognise long-term relations between inputs [24].

Some authors, however, have shown that cascaded DLRCs bring little to no benefit with more than two stages, where each DLRC following the first one only has access to the latter's output [24]. This, however, no longer remains the case for designs where the raw input is combined with the previous output at every stage. This way, a cascade of DLRCs will behave similarly to the Successive Approximation Algorithm used for Analogic-to-Digital Converters (ADCs), where the output gets more and more fine-tuned

at each stage. This was also shown to be of great value regarding classification tasks, as the RC has more opportunities to classify inputs accurately [11].

This method leads to an issue of its own because the standard methods to optimise each of the parameters for one DLRC are entirely unusable. An analysis to find the optimal solution of such a high-dimensional space is out of the question since it would be too demanding and time-consuming, as is the case for gradient descent solutions due to the higher probability of it being stuck on a local minimum, instead of the infimum of a cost function.[11] It might also be troublesome to find optimal values for all parameters when conflict occurs between pairs of them, leading to a compromise that will negatively impact our results. [11]

Despite this, the random optimisation (RO) method can mitigate this drawback by iteratively selecting the best parameters for each DLRC sequentially, as described in [11, 18] . On the other hand, setting a global input to each stage implies that the model is no longer a black-box system, as the intermediate processing is no longer completely unknown.

1.3 Background

As mentioned in 1.1, DLRCs belong to the family of RNNs, where the current depends on the network's previous states. Being the focal piece of our model, the feedback algorithm I.1 combines different information at a particular instance of the delay line to generate the next delay line value. That information comprises the masked input data, the output from the previous time step and the output computed for the previous input sample at the current time step. To better understand the predictive capability of the DLRC, it is best to look at how the aforementioned information is obtained while tracing the path followed by the input as it is processed. The original representation of a DLRC is present in I.

The first step is to mask the input by computing the product between the current input and the mask vector. Since this vector has the same length as the delay line, the result of this step is a masked sample for each delay step. Simply put, we would provide multiple perspectives on the same object to generate a clearer picture. In more rigorous terms, we perform a dimensionality expansion of the input to a high-order space where hyperplane separability is achieved. [9]. Hyperplane separability belongs to an integral part of convex set theory. [4] In short, ESNs should have enough structural inhomogeneity to exhibit a vast collection of complex dynamics that can be used for any task, similar to how some theories believe a biological brain works. One way to achieve this is through the choice of mask values. However, there is little incentive to optimise the mask values themselves as all that is required from these is to be different enough to generate “rich dynamics”. It is also critical that the same mask vector is applied to each input, or else the correlation between different inputs at a given delay step is lost. After masking the sample, the next step is to apply the input scaling hyperparameter, γ , which determines the portion of the nonlinear function used. This parameter, together with the remaining two

hyperparameters, α and η , explained further below, are configuration variables external to the model to be manually tuned or optimised for different tasks. Concerning the feedback algorithm, consider a delay period of τ and the number of neurons in the delay line as N_T for notation purposes. Let us refer to each tap of the delay line as a “virtual neuron”, V_k , where k represents the number of the delay step. In addition, suppose that the value held by a given virtual neuron in the previous delay line at instance k is P_k and that the current masked input sample is S_k . We will refer to P_k as the “parent value”. This feedback requires the contribution of two hyperparameters: α , which represents the filter factor, and η , which expresses the feedback learning rate. Both hyperparameters and γ should be within $[0, 1]$. Lastly, assume the nonlinearity node to be represented by f_{NL} .

$$V_k = V_{k-1} \cdot \alpha + (1 - \alpha) \cdot f_{NL}(S_k + P_k \eta) \quad (1.1)$$

$$k \in [0, N_T - 1] \quad (1.2)$$

In 1.1 results from applying backward Euler discretisation, a basic numerical integration method for ordinary differential equations, to a delay differential equation used to describe the continuous-time behaviour of a reservoir computer, as reported in [11]. A schematic of this filter is shown in Annex 1.1. Breaking down the equation by focusing on each hyperparameter gives us a simple perspective over the nonlinear dynamic system. Firstly, the filter factor splits the computation into two major components: the value of the previous output in the current delay line and the nonlinearity output. The larger the value of α , the more the memory effect from previous virtual neurons lasts, and the less impactful the value from previous inputs is. As stated in [11], given that the filter only possesses one pole at $z = \alpha$, it remains stable for as long as $|\alpha| < 1$. The memory effect aforementioned is related to the fundamental echo state property described in [9], “fading memory”. Simply put, as new inputs are processed, the imprint of older inputs is progressively eroded until they become wholly negligible. The values of hyperparameters, such as the filter factor, define the existence of echo states. For that matter, and because optimisation of these parameters is not the aim of our project, the values chosen will be based on literature, thus assuring the echo state property.

ESNs are inherently nonlinear dynamical systems due to their recurrent nature, and thus, using a linearity would not be adequate. One way to visualise that is through “cobweb plots.” These are iterated maps of one-dimensional functions starting from an initial condition. In A.3, we can see that infinity works as an attracting fixed point in the linear map, repelling every trajectory from the origin. Because these trajectories never return, it is impossible to generate intricate dynamics such as those in the 3rd-order polynomial map. as shown in A.4 and A.5 [23]

For this reason, the nonlinearity function is the main responsible for the dynamics of this system. The nonlinearity maps its input through a “bump function”, generally described below.

$$y(x) = \begin{cases} f_{NL} & \text{if } 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

The choice of nonlinearity is hard to optimise intuitively, as the result is a complicated nonlinear transformation of the input time series into an N high-dimensional output time series.

As a byproduct of 1.1, a state matrix $j \times k$ is obtained, where j is the number of input samples. During supervised training, what follows is the computation of weights by minimising a loss function over the state matrix. Given that RC predictions result from linear regression, the least-squares method is the most adequate due to its simplicity. For input X_i , the obtained predictions are:

$$Pred_i(V_k(x), W_k) = \sum_{K=0}^{k=N_T} W_k \cdot V_k(X_i) \quad (1.3)$$

In this method, w_k weights are adjusted to minimise the sum of squared residuals between each predicted value, $Pred_i$, and its target, T_i . The loss function is shown in (eqs. 4 and 5).

$$\begin{aligned} r_i &= T_i - Out_i, \\ \chi &= (T_i, X_i), \\ L(w|Chi) &= \sum_{i=0}^{N_{inputs}} r_i^2 \end{aligned} \quad (1.4)$$

During training, the model can become over-optimised for the training set and thus less capable of generalising. This is called overfitting. Regularisation, such as Tikhonov's method, or L2, prevents overfitting by adding a term to the loss function that penalises exceedingly large weights. The strength of this penalisation is set by γ , a regularisation coefficient that must be carefully chosen. (6)

$$L(w|Chi) = \sum_{i=0}^{N_{inputs}} r_i^2 + \lambda \cdot \sum_{K=0}^{k=N_T-1} w_k^2, \quad (1.5)$$

The state matrix is rebuilt for the test set, and predictions are computed as shown in (Eq.3). Lastly, as the standard metric of choice to compare the predictions with the ground truth, we use the Normalised Root Mean Square Error (NRMSE). The lower the value, the more accurate the predictions are.

Conversely, the normalised mean squared error (NMSE) is used to evaluate performance.

MATERIALS AND METHODS

2.1 General Scope

Our approach to designing a delay-line reservoir computer in hardware began by developing a higher-level implementation of the model in software, allowing us to ensure that the hardware was performing as intended. For this reason, based on an earlier Python prototype [10], our implementation had to be built as close as possible to the hardware.

In this approach, studying the memory costs of using different number formats for storing each value was critical. These include hyperparameters (α, γ, η) , mask values, delay line values, weights, outputs, and all intermediate results. A rough analysis of this subject can be seen in our results.

We aimed to take advantage of BRAM resources to store the weights and delay line values, sparing logic cells for the remaining memory requirements. Moreover, we must reserve a substantial amount of memory for other purposes, such as the microprocessor itself and its own memory space. To add to our rough estimate, the costs of implementation of a MicroBlaze softcore IP from Xilinx may take up to 7k logic cells, not including the BRAM reserved for the cache. This is particularly important given that we intend to work towards a cascade of DLRCs. Hence, we must be cautious with the resource usage per stage.

2.2 Materials

The goal is to deploy this model on a field programmable gate array (FPGA), an integrated circuit designed to be reconfigured. While Application Specific Integrated Circuits (ASICs) are ideal for deploying a finished application on a large scale, FPGAs are the perfect fit for the development phase as their physical structure can be rearranged post-manufacturing.

The FPGA/SoC board of choice for this project is the Arty A7-100T, featuring the XC7A100TCSG324-1 chip. This board possesses 101,240 logic cells, 240 DSP48E1 slices and 135 RAM blocks of 36 Kilobits.

We must use hardware descriptive languages (HDLs) to define our accelerator's

behaviour and structure such that tools can synthesise and implement it. Our language of choice was VHDL, and the set of tools utilised for design was Xilinx’s Vivado and Vitis, both v.2022.2. Briefly, Vivado was used to develop the hardware in VHDL, and Vitis to write an application in Embedded C. From the set of libraries included in the latter we used the library **lwip211** (v1.8) (Dunkels, 2003; Dunkels Woestenber, n.d.) and the echo template application to serve as a mould for the ethernet communication between the PC and the accelerator. Modifications were carried out to address a bug in the **lwip211** module. The revised code is detailed in [17]. As the tool to listen to Universal Serial Bus (USB) ports, “Putty” was the Secure Shell (SSH) and Telnet client of choice. To perform training and testing in software, we have used the **Numpy** (v.1.24.3), **Sklearn** (v1.3.0), and **Scipy** (v1.11.1) libraries in Python. AXI is a standardised interface protocol defined by ARM to establish communication between a microprocessor and peripherals.

RESULTS

3.1 Experimental Setup

3.1.1 Python Prototype

To build the model described in 1.3, we avoided the usage of external libraries to ensure that we had total control over the model to later implement in hardware. The first step was to build a DLRC as a class, an object, including instance methods for the nonlinearity, feedback loop, training, and testing functions.

In the preparation function, the mask values are read from an external file containing a set of six values $\pm 0.2, 0.6, 1$ uniformly distributed. Scaling, when performed, was achieved by using the min-max scaling function from **Sklearn** to a range $[-1, 1]$ to match that of the nonlinearity [20].

Once computed, the output matrix undergoes post-processing, where a column of ones is appended to implement the bias as an extra virtual neuron per delay line. To compute the weights, we use the “`linalg.lstsq()`” function from **Scipy**’s library to implement the least-squares algorithm with regularisation (Scipy (v.1.11.3) [22]).

After being trained, the model uses the test set to generate a new output matrix so that we obtain predictions using the trained weights. Finally, we measure the NRMSE between the weighted sum and the ground truth of the test set for each input sample.

Two Python scripts were built for each benchmark test, one with floating-point numbers, FP, and another with fixed-point numbers, FXP. For the latter, numbers were multiplied by $2^{N_{bits}}$ and then passed through a “`numpy.floor()`” function to remove the exceeding fractional component for a given precision. The number can be reverted to its FP format by multiplying by $2^{-N_{bits}}$.

Consider the number 0.375 and an FXP representation with 2 bits, with the fixed point on the leftmost position. Given that the LSB value is 0.250, it is evident that the exceeding fractional component would be lost in this representation. With the integer approach, 0.375 equals 1.5, and flooring yields 1, resulting in 0.250 when converted back to FP. This loss of precision is unavoidable in hardware and must be accounted for in our high-level prototypes. Lastly, we assessed the impact of such loss in precision for each benchmark

within a range of FXP representations.

3.1.2 Benchmarks

3.1.2.1 Overview

To ensure that our model was performing as intended, we opted for three distinct benchmark tests: k-step-ahead forecast (KAF) of the Lorentz system, one-step ahead forecast (OAF) of the Santa Fe laser time series, and, lastly, a system identification task known as NARMA10. This set of tests provides good coverage over a spectrum of nonlinearity complexity.

3.1.2.2 Lorentz Strange Attractor

The Lorentz system is a well-known ordinary differential equation in the field of mathematics that was derived from a model of airflow dynamics in the atmosphere. This deterministic nonlinear system can generate chaotic sequences, depending on the parameters selected, while remaining always bounded. This is the most basic description of an attractor, and part of the reason we chose this benchmark test is the high sensitivity to initial conditions that make it a strange attractor [9]. Moreover, predicting $k > 1$ step is more demanding than just one step and, thus, a good test for the sensitivity of the DLRC to hyperparameter changes.

The Lorentz Attractor is described by the following system of differential equations:

$$\frac{dx}{dt} = \sigma(y - x) \quad (3.1)$$

$$\frac{dy}{dt} = x(\rho - z) - y \quad (3.2)$$

$$\frac{dz}{dt} = xy - \beta z \quad (3.3)$$

$$[\sigma, \rho, \beta] = [10, 28, \frac{8}{3}]$$

We have generated a data set by solving the Lorentz system described up to 100 setpoints with $dt = 0.01$, generating 10k data points from two different initial conditions.:

$$A1(x; y; z) = (1; 1; 1) \quad (3.4)$$

$$A2(x; y; z) = (0.5; 2; 0.6)$$

It is common for recurrent neural networks to require a “warming up” period before stabilising; for that matter, the first 100 outputs were discarded.

Concerning the training and testing sets, we split the data set in half for each and did two experiments: training and testing on the data generated from the same initial condition, training on the data from attractor A1, and testing on data from attractor A2.

This double experiment is intended to assess the ability of our model to generalise for different initial conditions.

We have chosen a regularisation constant of $1 \cdot 10^{-5}$ for this benchmark. Other than this, the output matrix computations for each of the three variables were performed separately with $\eta = 0.3$, $\alpha = 0.9$, a min-max scaling between $[-1, 1]$ and with the third order polynomial as nonlinearity. The metric used to evaluate performance was the root normalised mean-square error. (NRMSE).

3.1.2.3 Santa Fe Chaotic Time Series

The second benchmark, the Santa Fe laser time series, originated from a competition at the Santa Fe Institute, New Mexico 1991, where cutting-edge models for time series predictions were evaluated. The participants were given datasets generated from lasers operating in a chaotic regime.

We followed the approach in [10] for our benchmark and used the second dataset “A_norm.cont” [26]. This dataset contains 10k data points from intensity variations of a pumped NH-3 laser operating in the chaotic regime, and the goal was to predict one sample ahead of the provided input sample.

Training and testing were conducted with 3k samples each, using $\eta = 0.4$, $\alpha = 0.7$ and a min-max scaling between $[-1, 1]$. We used the 3rd-order polynomial as nonlinearity; no noise was added to the output matrix, and the regularisation factor was 0.01. The warming-up period was up to 300 outputs, and the evaluation metric for this benchmark was the normalised mean-square error (NMSE).

3.1.2.4 Narma10 System Identification

Lastly, the NARMA10 system, represented by (8), is a discrete system that takes randomly generated inputs to generate a highly nonlinear dynamic response. The dynamic response from the NARMA10 system is deterministic for a given seed of randomly generated inputs. This benchmark test aims to feed the same noisy inputs to the model and train it to replicate the same response of the NARMA10 system. This response is a challenging experiment to perform on RCs due to the dependence on inputs of up to nine steps in the past, exceptionally demanding on the model’s memory.

$$Y(k+1) = 0.3Y(k) + 0.05Y(k) \cdot \left[\sum_{i=0}^9 Y(k-i) \right] + 1.5U(k-9)U(k) + 0.1, \quad (3.5)$$

$$f_{nl} = x(1 - \rho + \rho \cdot x)$$

The inputs were generated from a uniform distribution between $[0, 0.5]$ and the targets for a given discrete step, k , from (8), which was projected one step behind to create a spatial gap between the most recent input and target. Likewise, because the noise input is highly compact, we used time embedding to allow the model to relate samples in such

high nonlinearity. This was done by masking both the current and previous input samples together.

Training and testing phases were carried out using 10k samples each, using $\eta = 0.3$, $\alpha = 0.9$, $\gamma = 0.15$. The chosen nonlinearity in this case is (9) with $p=0.7$. Due to the exigency of memory, a weaker nonlinearity is called for, as described in [10] [19]. Lastly, no noise was added to the output matrix, a warmup period of 1000 samples was considered, and the prediction metric was once more the NRMSE.

$$NRMSE = \frac{\sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}}{\sigma_{Target}} \quad (3.6)$$

3.1.3 Hardware prototype

3.1.3.1 System Overview

The SoC architecture in our design comprises a MicroBlaze soft processor, a central processing unit (CPU) that manages data transfers and controls every peripheral in the design. We have configured the CPU for 32-bit numbers, using signed 16-bit FXP numbers for most signals in the design. To transfer data between the FPGA and a PC, we have chosen a standard set of IPs for the peripherals, including an AXI Uartlite and an AXI Ethernetlite (See Fig.A.1).

Since we intend to develop a Hardware-in-the-loop (HIL) system, we must establish communication between another machine and the hardware. We included peripherals such as an AXI UART Lite and an AXI Ethernet Lite for communication (See Fig.A.1).

This design uses a clock with a frequency of 83.6667 MHz, generated by the Memory Interface Generator, as recommended by Digilent's guide. The AXI Ethernet Lite is the only exception, driven by a slower clock with 25 MHz.

We exported the hardware design above to Vitis to ensure the ethernet and UART communication functionalities were operational and built an echo-state application using a Xilinx template. This application reproduces whichever input is sent to the network interface, echoing it back, and no change was made to the template.

To design the DLRC itself, a custom IP was packaged from a collective of components: a controller, a mask generator, a local BRAM and the DLRC core (See Fig. A.2).

The system communicates with the core and the AXI Interconnect, responsible for routing the signals from every IP to the CPU. For this reason, only the DLRC core has an AXI interface, and thus, it shares the same address space with the remaining components. The central core has to manage this shared address space, and for that matter, it contains two processes that decode the address being sent from the microprocessor, either an address to read from, AXI_ARADDR, or as an address to write to, AXI_AWADDR B.2.

It should be highlighted that data sent to other entities is not synced with the clock on the central core but rather on the entity that receives it to prevent timing mismatches.

Secondly, these processes must be triggered when the microcontroller requests a read or write operation from the application while the DLRC processes inputs.

As a preprocessing step, the two least significant bits must be removed from the address. Removing these bits is standard for addressing 32-bit memories as each address should map to four bytes of memory (32 bits), also known as memory structure alignment.

Regarding the writing address manager, it was essential to make sure that we could write weights to BRAM while the DLRC was idle and also to write the values for each virtual neuron when it was operating.

This is conflicting because, in each case, the signals driving the address come from different sources, and only one can be active at all times. Weights are sent to an address controlled by the application, while delay taps are written to an address generated by the controller. More details in the subsequent sections will clarify this.

As a workaround, we implemented a register, `axi_switch_awaddr`, that alternates between both address sources depending on whether the start signal is high or low. If low, the DLRC is inactive, and thus, only the addresses from the application are considered. If high, then only the addresses from the controller are sent to the BRAM.

It should be noted that this was only the case for addresses within the space reserved for the BRAM. Local registers were still accessible to writing actions from the CPU regardless of the current state of the DLRC. These include configurable registers for the parameters α and η , control registers `src_start` and `src_stop`; and lastly, the input register. Moreover, there are read-only status registers to monitor the system's state.

The design for the reading address manager is similar, as we need to read values from the BRAM using addresses from the controller (e.g. to get the parent value at an instance k) but also from the application. To interface the address space of the DLRC IP with the application, we have used a structure mapped to the base address of the IP (See Fig.B.1).

Other components access the internal registers of the central core via the output ports, which are synced with these.

3.1.3.2 Block of Random Access Memory

To synthesise a BRAM block, we have chosen the inference method, which consists of adding an entity using Xilinx's recommended coding examples. (Xilinx Inc, 2022) The BRAM block best suited to our design was a Byte Write Enable (BWE) True Dual Port (TDP) BRAM in write-first mode. This is because we can select which bytes within a memory address we write to, and thus, we can write the delay taps in the same address as their respective weight. Our BRAM is split into two columns, each 16-bit wide and controlled by a write enable signal (See Fig. 3.1).

The main advantage of our approach is that each 32-bit word consists of a pair of weight and delay values tied to the same memory address. With this strategy, we can effectively loop over memory, using a single signal from the controller as the address, "`SCT_CNT_ADDR`". Additionally, this design allows us to take advantage of using a

smaller number format to store two distinct values in a single address, further improving resource usage.

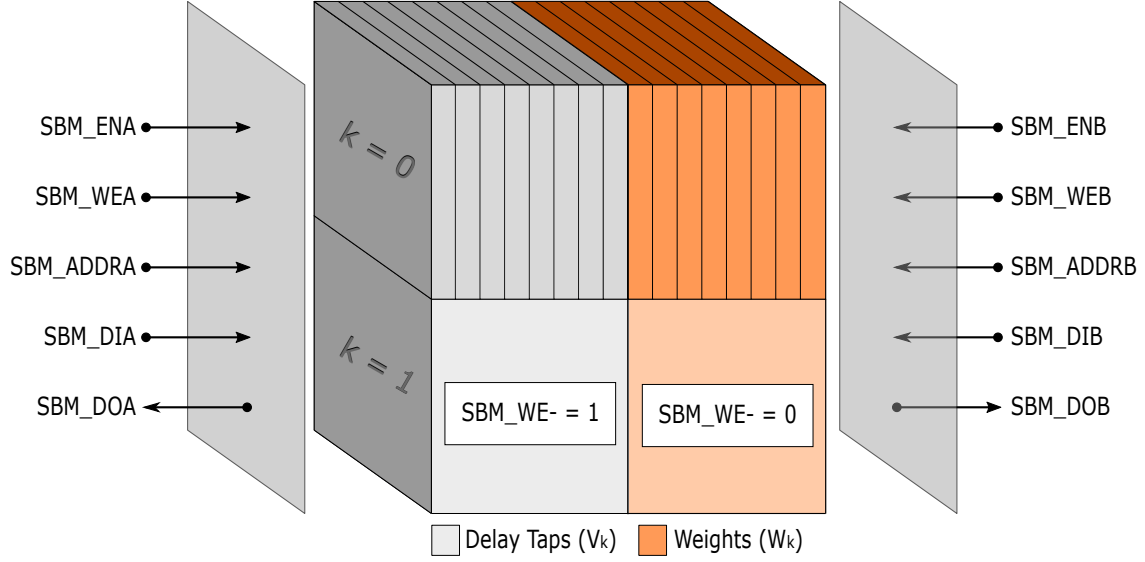


Figure 3.1: Bram Structural view. Note: Each row has 16 columns instead of 32 for simplicity of design

As the DLRC computes the delay taps, it writes those values to BRAM, as shown in B.2 Finally, as shown in B.3, the BRAM outputs delay taps and weights correctly.

3.1.3.3 System controller

The controller consists of a Mealy-State Machine (STM) that operates between three distinct states: IDLE, SET and IN_LOOP (See Fig. 3.2). Outputs from Mealy STMs depend on inputs and the system's current state, tying outputs to transitions instead of states. Since states require encoding, which in turn costs memory, having outputs on transitions opens the possibility of using fewer states, thus consuming less memory.

Despite this, our STM cannot be said to be pure of the Mealy type as one of the outputs: the address counter is generated under a synchronous process instead of the remaining outputs, which are generated asynchronously. The latter are generated under one process with a sensitivity list exclusively constituted by inputs determining the next state. At each rising edge, the next state becomes the current state, and the address counter is increased until the whole delay line is covered. The loop continues seamlessly until no more data is loaded. (See Fig.B.5)

In our controller, state transitions and the address increment occur once every two clock cycles. The main driving force for the existence of this delay circuit is the feedback loop. Because we are using a pipeline, in the absence of a delay circuit, the current_value had to be updated, and simultaneously assigned to previous_value. Because current_value is forcefully a signal, it needs one extra clock cycle after being assigned a value to receive it.

Thus, it is inherently impossible to update the previous values, until the next rising edge, which would coincide with the computation of the next `current_value` in the pipeline. By extending the period of each computation to two clock cycles we are then able to update the `current_value`, and pass it onto the `previous_value` in due time B.18. Additionally, this allows us to simplify computations such as the nonlinear node, computing part of the result in one delay, and the remaining in the other B.17.

As can be seen in 3.2, there are three main output signals: a ready signal that would sync the input sample with the output provided by the previous stage, A start signal that controls the flow of the mask generator and the pipeline, And, lastly, a done signal that would be used to forward the prediction to the next stage in the cascade. However, as it will be further discussed in 3.2.2.2, it was not possible to implement the cascade in its entirety, and for that reason, the “`src_ready`” and “`src_done`” signals did not fulfil the purposes above.

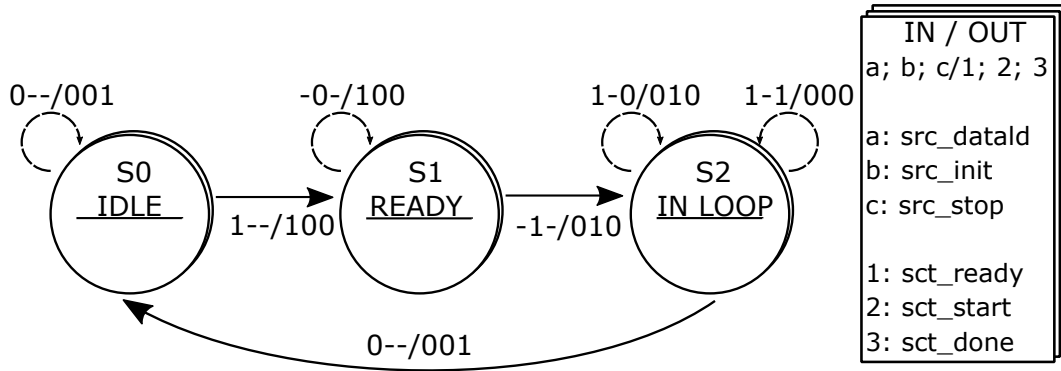


Figure 3.2: State Machine diagram for the implemented controller. Note: ‘-’ denotes a “don’t care”.

3.1.3.4 Mask generator

The mask generator aims to send one out of a set of mask values at each new address received from the controller. The set of mask values is stored in an array of registers, similar to that of the BRAM, but smaller. These registers can be written to and read from exactly like the BRAM. Our design philosophy focused on the efficient use of resources, and the design for this component was no different. To avoid storing every mask value, which could get expensive for longer delay lines, we generate 5-bit addresses, theoretically uniformly distributed, that are used to pull mask values from the 32-deep memory structure. In more detail, we have chosen the logistic map in a chaotic regime with $r = 4$ to generate 31-bit numbers, from which we extract the last 5 bits to use as addresses (See Fig.3.3).

Ideally, the address distribution should be as uniform as possible so that each mask value has an equal chance of being used. One additional note is that values must have

enough precision so that the addresses extracted do not repeat one after the other, as that would be an obstacle to attaining rich dynamics in the reservoir [9]. In the case of digital numbers, there is a finite number of possible values, and therefore, the less precision there is, the more often replicas will appear. Our results on this matter will be discussed in the next chapter 3.2.2.1.

Lastly, we had to compute each mask value in advance to ensure that the instance at which the central core receives the first mask value matches the start of the pipeline operation. For instance, the first 5-LSB result is not computed by the logistic map but rather initialised at the beginning of every delay line. The second result is computed instead. This was only possible by having a constant initial value for the mapping function. Hence, we know ahead of time the exact output of the first iteration of the logistic map.

As an added benefit, this implies that we skip the computation of one address value per input sample. More importantly, instead of solely writing each mask value in the set of 32 registers through the embedded application, we write its product with γ , saving more resources. Intuitively, this yields the same as scaling the input first and applying the mask at last.

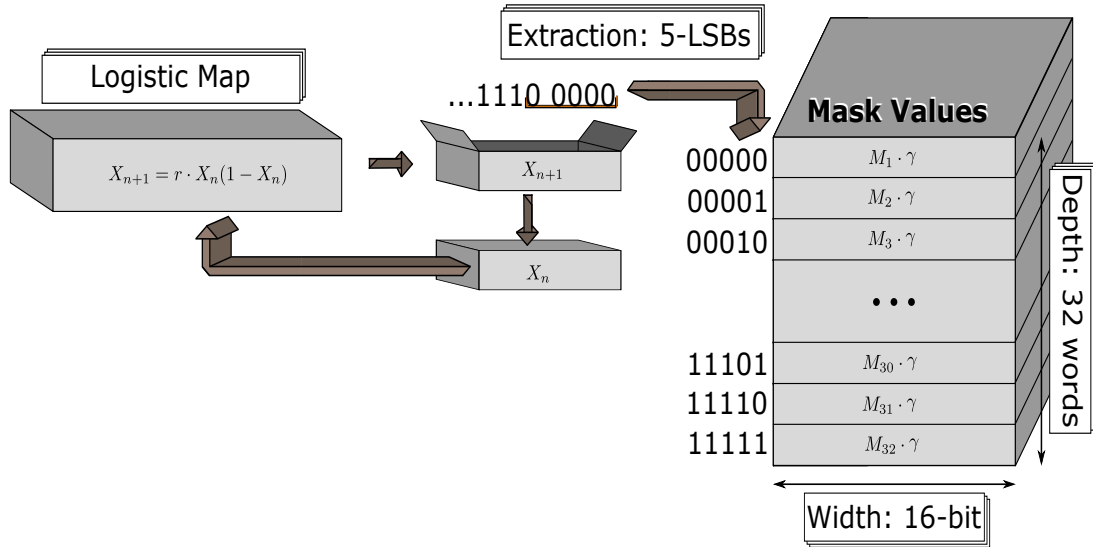


Figure 3.3: Mask generator flow diagram. 5-LSB extraction from a number generated through the logistic map yields an address that is used to output the corresponding mask value.

3.1.3.5 Pipeline design

Having explained the functionality of every other component of the DLRC IP, we must now clarify how everything fits together. As mentioned in the beginning, the central piece of this IP is the main core.

The pipeline is built from a custom AXI_LITE [17] interface template from Xilinx, to which we included a double-input buffer, a simple delay circuit, the stages of the pipeline and the changes to the writing and reading address managers mentioned in 3.1.3.2.

To be more precise, our implementation is different from a typical pipeline. The time to compute the total of delay taps per input is constant. Despite this, the delay taps within one delay line are pipelined, in the sense that it takes ten clock cycles to produce the first delay tap and only one for those that follow. Furthermore, the pipeline implemented is not strictly linear, as some stages operate simultaneously on different inputs.

Double buffering allows our design to continuously process the current input sample while transferring the next into a local register. Consequently, the pipeline does not stall after each input is processed as the following sample is already available, increasing throughput. (See B.10)

In our design, we included two status signals: one to monitor if the buffer has a vacancy, `st_vacant`, and the other to trigger the end of the computation, `st_finished`. `St_vacant` is kept raised as soon as the last input in the buffer starts being processed; thus, `st_finished` is triggered if said vacancy is not occupied by the end of the current computation. If so, the controller then reads the status of the pipeline, halting its operations B.5, B.11. While the pipeline is busy, the CPU checks for the state of vacancies in the buffer, sending an input once the respective status signal is raised. To collect the outputs, the CPU checks the finish status signal and reads the values from BRAM. Ultimately, this method proved ineffective when running the embedded application, as will be further discussed in the next chapter.

As mentioned in 3.1.3.2, we intend to use the addresses generated by the controller to loop over the delay line, storing each output accordingly. Since each delay tap is computed over multiple stages and thus delayed, it is critical to ensure that the addresses generated by the controller remain paired with their respective delay taps. To achieve that, we have implemented a simple chain of ten registers so that addresses and output values were delayed for the same amount of clock cycles by the end B.12.

Additionally, because one clock cycle per stage may cause overhead in some instances, each generated address and mask value was extended to 2 clock cycles using two toggle registers as a delay circuit. This delay circuit prevents timing violations in computation-intensive stages such as the nonlinearity stage. It is also a simple and effective method to implement the feedback loop in a pipeline.

The first stage of the pipeline collects and syncs each mask value to the input sample and the previous stage's output. However, as will be further discussed in the next section, we have only implemented a single-stage DLRC, and thus, this first stage acts solely as foundational work towards a cascaded DLRC. Masking of the input sample occurs in the second stage B.13.

In the third stage, weights and delay values are retrieved from the BRAM and passed onto the pipeline. The learning rate is also applied to the delay value B.14. It is important to note that it takes one clock cycle to assign the read value to the output port, `SRC_DOB`,

and another to apply the learning rate. Whereas the masked input only takes one clock cycle. Because the outputs of Stages 2 and 3 must be synced, we use `ppl_addr1` as the reading address, effectively reducing the delay to a one-clock cycle. The masked input and the parent value are then combined in stage 4 as an input to the nonlinearity B.16. We have implemented the same 3rd-order polynomial for the nonlinearity used for both the Lorentz and Santa Fe benchmarks. In this stage, we used the delay circuit to reduce the intensity of computations per clock cycle. In greater detail, we compute the input squared and the following subtraction during the first delay. For the second delay, we computed the product between the multiplier and the input, and lastly, the output of the nonlinearity B.17.

Finally, as the last step of the pipeline, the filter factors are applied to the current and previous values, which are then added together, precisely as explained in 1.3. In this final stage, we have once more capitalised on the delay circuit to first compute the current value and then update the previous value in the next clock cycle B.18. This approach is synthesisable and implementable in hardware without critical warnings or errors.

3.2 Discussion

In this chapter, we will review the results of our experimental setup, commencing by analysing the performance on the aforementioned benchmarks, discussing potential oversights and commenting on the impact of precision, or lack thereof. When we mention a 16-bit FXP format, we mean 15 bits of precision and one bit for the sign. Further in this section, we will address some of the limitations of our hardware implementation, together with possible solutions. Lastly, we will comment on resource usage, performance, speed, and accuracy compared to our software implementation.

3.2.1 Benchmarks

3.2.1.1 Lorentz Strange Attractor - Discussion

The first task we submitted our prototype to was OAF forecasting, which was then increased to multiple steps ahead. We have limited the number of virtual neurons to 256 and the number format to floating and 16-bit FXP. The separate predictions for each variable were compared to their target values, obtaining the following attractors. (See Fig.C.1 and Fig.C.2)

The results are that we achieve an acceptable representation of the attractor for both number formats. Despite the visible differences, it is difficult to say which is closest to the original attractor and which variable is the most inaccurate. That can be seen by the regression plots (C.5, C.6) and the time series plots (Annex 3. X).

Below is a table comparing our results with some of the best models currently in OSA Forecasting for the dynamical equation of X.

Table 3.1: One Step Ahead Forecasting for $X(t)$. Comparison with Multi-stage deep ESNs [19].

Lorentz Attractor $X(t)$ One-Step ahead prediction		
Model	NRMSE	Runtime
HESN-ARF	0.000046	75.89s
DLRC-256	0.001094	11.33s
DeepESN	0.000059	38.63s
DBEN	0.000068	14.46s
ADRC	0.000057	40.25s
Naive	0.055400	-

Although our results may not appear as impressive as other approaches, we must remember that this is a single DLRC compared to cascades of larger networks. [19] Furthermore, we have not optimised the parameters for this benchmark as it was outside of the scope of this project. Likewise, this will be the only comparison with other models in regards to performance, simply because we are not expecting the best results in that regard.

To limit-test our model, we have performed KAF for K values up to 9 steps ahead. The results are shown in (C.8, C.9, C.10)

Lastly, we tested the FXP prototype over a range of precision bits to evaluate the influence of number formatting in this benchmark. Results are displayed in C.7

By analysing the results above, particularly C.7, it seems that the Z variable is the hardest to predict and the one that benefits the most from the increase in precision. We believe the reason for this is tied to the dependencies of each one of the differential equations in the Lorentz system. For instance, the X variable depends on itself and the Y variable but not the Z variable. In comparison, the Y and Z variables depend on all three system variables. Moreover, the likely reason Z is more challenging to predict compared to Y could be due to an increased nonlinear effect, which is evident in the time series. In other words, Z has a more aggressive curvature with steep oscillations. C.3

However, it is interesting that after 8 bits of precision, performance matches that of the FP number format for every variable, revealing that precision can be limited without significant drawbacks for this task. This observed behaviour can be due to an overlook in the model on our end, thus hindering its performance. Or, perhaps it is an inherent limitation of this model when predicting strange attractors.

Moreover, as demonstrated by the results of (C.8, C.9, C.10), increasing the number of steps to predict significantly increases the measured NRMSE. Strange attractors such as the Lorentz Attractor are highly sensitive to initial conditions, and thus, the slightest of deviations generate entirely different trajectories. By increasing the number of steps ahead, we give a margin for such errors to occur and escalate, worsening predictions.

We have also experimented with testing the model with data originating from a different initial condition than the one that generated the training set and got comparable

results to what is shown in (C.11, C.12, C.13). These results prove that our model can generalise for Lorentz attractors with different initial conditions.

3.2.1.2 Santa Fe Time Series - Discussion

In a similar fashion to the previous benchmark, we will compare the predictions of both the 16-bit FXP and the floating number format as a time series (C.14, C.15) and as a regression plot (C.16, C.17). While the former allows us to see the dynamical system predicted through the notion of time, the latter hides that characteristic to focus exclusively on accuracy.

Lastly, continuing to assess the impact of precision on performance, we tested the FXP prototype over a range of precisions.C.18

The results are once again within what was to be expected. Both predictions match the target reasonably, except for some overshoots and undershoots. However, comparing both time series makes it difficult to say which was more accurate and by how much. That is where our regression plots can assist.

Looking at C.16, C.17, we can see that the floating number format is preferable over the FXP format, but not to the extent that would discourage us from using the latter to perform tasks of similar complexity. Hence, the 16-bit FXP continues to stand as a great option to be implemented in hardware.

3.2.1.3 Narma10 - Discussion

As explained in our methodology, significant changes were made to the preprocessing. For instance, time-embedding and a weaker nonlinearity were included in our model. Both improved dynamical memory while maintaining a good level of nonlinearity. As widely stated in the literature, nonlinearity and dynamical memory act as opposing forces, and thus, the improvement of one generally implies sacrificing the other to some degree. This strategy proved effective for the Narma10 series, as shown in the following results.

Firstly, we assessed the model's accuracy using only 256 neurons for the floating and FXP number formats as displayed in C.21 and C.22. This time, given the complexity of the task, we lifted the limitation of using a maximum of 256 virtual neurons. This way, we got an insight into how many neurons would be needed for the most complex tasks. For the time series of each, see C.25 and C.26

As usual, we assess the importance of precision for this task to conclude our results from the benchmarks. C.27

From this benchmark, it became clear that even with the precision of FP numbers, small networks will still struggle to perform well in highly memory-demanding tasks such as the Narma10. This favours our implementation because we face fewer restrictions when implementing more virtual neurons. In our case, high dimensionality emerges from the time domain, whereas other approaches need to implement increasingly more resources and complex interconnections to accommodate larger networks.

However, by increasing the number of delay taps from 256 to 800 per delay line, we increase the number of weights, and as a consequence, we raise the leverage for fine-tuning and further specialising neurons.

Additionally, as we can see from C.23 and C.24, it is possible to obtain good results by increasing the network size. However, we must pay attention to the disparity in performance improvement between both number formats. Consider the improvement metric (eq.7), where N_0 is the lower number of neurons, 256, and N_1 is the largest, 800.

$$Improvement(N_0, N_1) = \frac{|NRMSE(N_1) - NRMSE(N_0)|}{NRMSE(N_0)} \cdot 100\% \quad (3.7)$$

The improvement metric shows that increasing the number of neurons from 256 to 800 is more significant for higher precision. Namely, we reduce the error by 43.82% for the FP number format, while for the FXP number format, NRMSE is only decreased by 15.26 %.

This has severe implications for our hardware implementation as it may not be worth the extra delay steps over more compact networks. More work should be done towards exploring broader ranges of precision and their impact on performance.

3.2.2 Hardware prototype - Discussion

3.2.2.1 Mask Generator

Our design for the mask generator was motivated by the goal of being as efficient as possible using memory. Because each stage must have as many mask values as weights or delay taps, storing every mask value would effectively increase BRAM usage by 50%. Instead, we have opted to store exclusively each unique value of the mask. As long as the mask values are still randomly distributed across the delay line, this is a highly efficient way of storing the mask values.

There are multiple options for random number generators in hardware. However, the obstacles we have faced with most of these were two-fold: for one, it would be difficult, if not impossible, to implement the same generator in both software and hardware. Moreover, since most generators are stochastic, ensuring that the same seed was used in both environments was unlikely.

For this reason, we sought a non-stochastic solution that would be reasonably simple to implement in both environments. The logistic map in a chaotic regime fits these requirements quite well. As an added benefit, it allowed us to simulate and debug easily, as the mask values were consistent on every run.

By virtue of these benefits, we have developed a Python script that matched the exact behaviour in hardware and tested for different initial conditions and bit precisions. This is an essential step because applying a map to a digital number of fixed precision will likely produce "replicas", copies of a value in subsequent instances. The lower the precision, the smaller the pool of numbers that the map has access to, and thus, the more likely it is to yield replicas.

For example, D.1 shows the result of generating 8-bit numbers with the logistic map.

As explained in the 1.3, RCs require structural inhomogeneity to exhibit a vast collection of complex dynamics. Replicas work like spacers and threaten this structural inhomogeneity in the sense that different input information is placed farther apart in the delay line. Over 1000 runs, on average, a replica was encountered every 15 delay steps of each 256-step long delay line. To produce up to 800 values without a single replica, a 25-bit number was required. Based on these results, we have chosen 31-bit signed FXP as the number format to be well above the requirements. This is unnecessary, as detecting one replica of a previous value after 15 steps does not threaten structural inhomogeneity. Future work may find a better balance between avoiding replicas with less numerical precision and further reducing implementation costs.

Despite the advantages above, our strategy limits the number of unique mask values to powers of 2. If that condition is not met, values will not be stored uniformly, and here lies the crux of this approach. If $2^{N_{bits}}$ addresses are produced with equal probability, and there is an imbalance with the frequency of each unique value, then mask values will not be uniformly distributed. For instance, if we would like to store a six-valued mask, four values will always be left out of the two remaining registers. We refer to this effect as the “offset bias,” which can be observed in D.2. However, we can use a more significant number of registers to reduce the discrepancy in frequency. Consider two distinct setups to accommodate a six-valued mask. Setup A uses eight registers, and B uses 32 and assumes a uniform discrete distribution for the population of addresses generated. Let M_k be the k th unique mask value. The probabilities of each mask value being used are the following:

To put it simply, the more registers there are, the more times each set of unique values is repeated. Independently of the number of registers, the offset bias only adds one more instance to the frequency of some mask values. Therefore, as the minimum frequency for every value increases, the additional instance becomes smaller in proportion to that, and thus the offset bias is lessened. To prove the effectiveness of this method, we analysed the distribution of mask values generated in hardware. D.3

Despite these results, we sought to investigate further to ensure our assumptions were correct. We assumed the addresses generated were random, but that was not observed after analysing their distribution with the 5-LSB extraction method. D.5

In [3], the logistic map successfully passed all 15 NIST tests for sequences between 100 and 1,000,000 bits. To formally test the uniform distribution of the addresses generated, we performed the “Frequency”, “Cumulative Sum”, and “Runs” from NIST. No more tests were performed because our datasets were restricted to a maximum of 256 samples, the delay line length. (See 3.2)

The NIST tests validate what we have just described: the addresses generated with the N-LSB methods were not uniformly distributed. After a certain number of extracted bits, the distribution returns to what is expected: a discrete uniform distribution. At a glance, this result could be due to using such few digits that we failed to collect a significant

Table 3.2: NIST: Set of statistical tests for discrete uniform distributions. Commonly used to validate Pseudo Random Bit Generators. On the rightmost column, there are 3 characters that correspond to the each of the three tests, F, CS and R respectively. "X" stands for failed the test for uniformity, while "Y" means that the test was successful. "Uniform" is the control. Tests performed at [21]

	P-Values			
	*1 forward and backwards failed/approved the same			
Datasets	Frequency (F)	Cumulative Sum (CS)*1	Runs ®	
5-LSB	0.00404	0.0054	0.910811	xyy
3-LSB	0.001778	0.002308	0.613839	xyy
20-LSB	0.017549	0.024839	0.537819	yyy
Uniform	0.0455	0.067173	0.703239	yyy

portion of the population of digits, skewing results. Nevertheless, we did not delve further into this subject due to time constrictions, so details remain to be investigated.

3.2.2.2 Embedded application interface

To build an application on top of hardware, our platform, we had to test the functionality of most peripherals such as the AXI Uartlite or AXI Ethernetlite, including the basic functionalities of the CPU (Ex.: Arithmetics, data transfers, among others).

For the UART, we built a simple application making use of “`xil_printf()`” to send data through the USB, and used Putty to listen to the port. Further tests were performed after implementing the BRAM to test simple arithmetics and ensure that the interface within the application and the FPGA were correctly done on both ends, whether the C structure in the former or the writing and reading managers in the latter. Communication through USB was successful throughout every test.

Conversely, the AXI Ethernetlite IP could not establish a network interface, hindering our progress and demanding a substantial investment of time and effort. Initially, we believed that the hardware was at fault. We developed multiple block designs attempting an intense trial-and-error approach to address this. However, despite our rigorous efforts, the problem persisted. Although optional, communication through Transmission Control Protocol (TCP) is highly desirable for real-time wireless applications.

After enough attempts, our suspicions grew towards the lwIP Echo Server template application files. Eventually, the source of the issue was traced to a file in the library of lwIp211 v1.8, namely “`xadapter.c`”. Two missing “`break`” statements within a “`switch...case`” construct caused the application to disregard a valid address, falling through both conditions into a default clause that exhibited the generic error message from previous attempts.

Once the network interface was established, a manually fixed IP was set for the FPGA to receive data and echo it back to the sender. From our later tests, the TCP connection was operational.

Table 3.3: Runtime comparison between software and hardware. Our design is at least 266 times faster than the fastest FP implementations.

Per 10k w/256VNs	
Prototype	time(s)
N10_float	1.63E+01
N10_int	7.16E+01
SF_float	1.69E+01
SF_int	6.70E+01
SoC_DLRC	6.12E-02

The application we aimed to build had two modes of operations: In the first iteration, the application would receive training data from a PC, send it to the DLRC for processing and collect the outputs of each input sample. These outputs had to be transferred to the PC to initiate training and compute weights. After that, the second mode of operations would commence: The Testing phase. Concisely, weights would be sent back to the FPGA together with testing data to compute predictions and operate in some HIL applications.

Unfortunately, our development was hindered right at the start of this plan owing to the inability to collect the outputs in due time. As previously mentioned, forcing the CPU to watch over a status signal, `st_vacant`, to collect predictions or to read the outputs in BRAM was not an adequate approach.

As the pipeline only stops when every input sample is processed, reading from the BRAM was not fast enough. To attest that, we implemented a counter that would increase by one every time an input sample was processed entirely. If our approach were suitable, we could see the counter increment by one at every instance that the `st_vacant` flag was read. We observed that the first two increments were accurate, but the ones that followed had a significant gap from the last counter value. (E.g. moving from 2 to 9 and 26).

Ultimately, this problem was left unsolved. There are, however, better alternatives to our first approach that could overcome this issue. For instance, implementing an interrupt after each input sample is fully processed forces the system to stall until the application has read every output value. Although adding an interrupt would affect speed, it is likely the simplest solution. Alternatively, an AXI_STREAM interface, instead of our AXI_LITE, could also be a good option as it is built for real-time data transfers.

3.2.2.3 Performance and Resource Utilization

Despite being unable to test our design in real-time hardware, the simulated results were substantially satisfactory. Our pipeline fully processed each input sample in 6,118 nanoseconds, equivalent to just under 165,000 unmasked input samples per second. This is an excellent step beyond what we could achieve in software; see 3.2.2.3.

Moreover, as our priority was to optimise resource usage, there is plenty of room for improvement regarding speed. For instance, in our block design, we have used the clock driving the MicroBlaze for most IPs, including the DLRC.

However, that does not have to be the case, as we can use the Clocking Wizard IP to generate clock periods down to 3000 ps or frequencies up to 333.33 MHz. [6] This, however, would imply altering the design to deal with clock domain crossing, avoiding metastable signals.

Another possibility is implementing a pipeline structure, where input samples would not stay buffered until the previous fully computed output. Instead, the mask values can be pipelined after being generated, similar to the addresses, and the input samples can be sent to the FPGA every two clock cycles in place of every ten.

These are some of the most evident changes to make, but multiple others must be explored.

Concerning resource usage, 3.4 exhibits the estimated values for a multitude of components in comparison to other highly efficient RNNs. ([12])

Table 3.4: Post-Implementation Resource estimates of the SoC single staged DLRC

Resource Type	In Utilization (DLRC)		Available	Utilization % (DLRC)	Utilization %
LUT	933	3440	63400	1.4716	5.426
LUTRAM	9	187	19000	0.0474	0.984
FF	1161	3635	126800	0.9156	2.867
BRAM	1	13	135	0.7407	9.630
DSP	11	11	240	4.5833	4.583
IO	-	21	210	-	10.000
MMCM	-	1	6	-	16.667

Lastly, and as our ultimate goal for this project, we have successfully implemented an exact twin of the software model in hardware. As shown in D.6, the outputs are incredibly close for the first four samples. The minimal differences between the two are primarily due to rounding rules, which are difficult to control and predict in hardware.

CONCLUSIONS AND FUTURE PERSPECTIVES

In the present day, concerns over each their own carbon footprint and energy consumption have become widespread. Renewable energy sources have started gaining traction over fossil fuels, and sustainable products and services have taken over the market. In general, we are transitioning from an Era where efficacy ruled over efficiency to the polar opposite.

However, for some technologies, as is the case for AI, the chapter of novelty has just begun to flourish, and with it comes a blind desire to see results. That is precisely how Large Language Models (LLMs) became such a staple in society, massive networks of immense computing power. Naturally, a few billion parameters have their fair share of energy consumption.

Despite all efforts, we are amidst a global energy crisis, and therefore, we should strive for efficiency and intelligent resource usage. [8] With this work, we intend to pave the way towards efficient computing algorithms, opening potential opportunities for exciting novelties. By showing the impact of precision over various benchmarks, we can make better use of memory. For instance, using more than 8 bits for the Lorentz map might not be as valuable as we thought.

We have successfully implemented a single-staged DLRC in hardware, which may appear modest in performance compared to more extensive neural networks but certainly makes up for the difference in efficiency for plenty of applications. We argue that our model has achieved better performance per bit of memory than those we had just compared. Further analysis would be required to prove or disprove this claim.

For our implementation of this machine learning paradigm, we have shown how a multitude of resource-efficient strategies can synergise together, driving down the number of resources and energy consumed from using the logistic map to generate addresses where only the unique values of a mask are physically stored, to taking advantage of the architecture of BWE-BRAM to make full use of memory in a controlled manner.

We have also shown that it is possible to implement this architecture of RNNs in hardware and that, despite its inherent time-multiplexing and consequent delays, speed was not an issue. In fact, their innate delayed nature matches exceptionally well with pipeline systems.

In the end, unfortunately, our ultimate goal of implementing a cascaded DLRC was not fulfilled. All this means is that there are plenty of options for future work. Given how efficient our implementation is, we are confident that a cascaded architecture may exhibit why efficiency can be just as fascinating, if not more in some instances, than efficacy.

A similar reflection can be had about the emergence of smartphones, ingenious pieces of technology that were only made possible by successive efforts of the microelectronics industry to miniaturise devices, reducing energy costs and improving performance altogether. Something similar is likely on the horizon for AI; Hopefully, it may be just as revolutionary.

BIBLIOGRAPHY

- [1] G. Aad et al. “Artificial Neural Networks on FPGAs for Real-Time Energy Reconstruction of the ATLAS LAr Calorimeters”. In: *Computing and Software for Big Science* 5.1 (2021-12). ISSN: 25102044. DOI: 10.1007/s41781-021-00066-y (cit. on p. 3).
- [2] L. Appeltant et al. “Information processing using a single dynamical node as complex system”. In: *Nature Communications* 2.1 (2011). ISSN: 20411723. DOI: 10.1038/ncomms1476 (cit. on pp. 2, 3, 78).
- [3] L. E. Bassham et al. *A statistical test suite for random and pseudorandom number generators for cryptographic applications*. Tech. rep. Gaithersburg, MD: National Institute of Standards and Technology, 2010. DOI: 10.6028/NIST.SP.800-22r1a (cit. on p. 22).
- [4] S. P. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge University Press, 2004, pp. 46–51. ISBN: 9780521833783 (cit. on p. 4).
- [5] A. X. M. Chang, B. Martini, and E. Culurciello. “Recurrent Neural Networks Hardware Implementation on FPGA”. In: (2015-11). URL: <http://arxiv.org/abs/1511.05552> (cit. on p. 3).
- [6] Digilent. “Arty A7 Reference Manual”. In: (). URL: <https://digilent.com/reference/programmable-logic/arty-a7/reference-manual> (cit. on p. 25).
- [7] K. Hao. *MIT Technology Review - Training a single AI model can emit as much carbon as five cars in their lifetimes*. 2019-06. URL: <https://www.technologyreview.com/2019/06/06/239031/training-a-single-ai-model-can-emit-as-much-carbon-as-five-cars-in-their-lifetimes/> (cit. on p. 2).
- [8] International Energy Agency. *Global Energy Crisis*. Accessed on 5th of September. 2023. URL: <https://www.iea.org/topics/global-energy-crisis> (cit. on p. 26).
- [9] H. Jaeger. *The “echo state” approach to analysing and training recurrent neural networks with an Erratum note 1*. Tech. rep. 2010 (cit. on pp. 1, 2, 4, 5, 10, 16).
- [10] L. Keuninckx. *Electronic systems as an experimental testbed to study nonlinear delay dynamics*. 2016. ISBN: 9789057185342. URL: www.vubpress.be (cit. on pp. 7, 11, 12).

- [11] L. Keuninckx, J. Danckaert, and G. Van der Sande. “Real-time Audio Processing with a Cascade of Discrete-Time Delay Line-Based Reservoir Computers”. In: *Cognitive Computation* 9.3 (2017-06), pp. 315–326. ISSN: 18669964. DOI: 10.1007/s12559-017-9457-5 (cit. on pp. 4, 5, 77).
- [12] K. Khalil et al. “Economic LSTM Approach for Recurrent Neural Networks”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 66.11 (2019-11), pp. 1885–1889. ISSN: 15583791. DOI: 10.1109/TCSII.2019.2924663 (cit. on pp. 2, 25).
- [13] B. Li et al. “Large scale recurrent neural network on GPU”. In: *Proceedings of the International Joint Conference on Neural Networks*. Institute of Electrical and Electronics Engineers Inc., 2014-09, pp. 4062–4069. ISBN: 9781479914845. DOI: 10.1109/IJCNN.2014.6889433 (cit. on p. 3).
- [14] S. Li et al. “FPGA acceleration of recurrent neural network based language model”. In: *Proceedings - 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015*. Institute of Electrical and Electronics Engineers Inc., 2015-07, pp. 111–118. ISBN: 9781479999699. DOI: 10.1109/FCCM.2015.50 (cit. on p. 3).
- [15] J. M. Lourenço. *The NOVAthesis L^AT_EX Template User’s Manual*. NOVA University Lisbon. 2021. URL: <https://github.com/joaomlourenco/novathesis/raw/main/template.pdf> (cit. on p. ii).
- [16] W. Maass, T. Natschläger, and H. Markram. *Real-Time Computing Without Stable States: A New Framework for Neural Computation Based on Perturbations*. Tech. rep. 2002, pp. 2531–2560 (cit. on pp. 1, 2).
- [17] R. Mafra. *DLRC Auxiliary GitHub Repository*. 2023 (cit. on pp. 8, 17).
- [18] J. Matyas. “Random Optimization.” In: *Automation and Remote Control* 1965. Vol. 26. Springer Science & Business Media, 1965, pp. 246–253 (cit. on p. 4).
- [19] X. Na et al. “Multistep-Ahead Chaotic Time Series Prediction Based on Hierarchical Echo State Network with Augmented Random Features”. In: *IEEE Transactions on Cognitive and Developmental Systems* 15.2 (2023-06), pp. 700–711. ISSN: 23798939. DOI: 10.1109/TCDS.2022.3176888 (cit. on pp. 12, 19).
- [20] F. Pedregosa FABIANPEDREGOSA et al. *Scikit-learn: Machine Learning in Python* Gaël Varoquaux Bertrand Thirion Vincent Dubourg Alexandre Passos PEDREGOSA, VAROQUAUX, GRAMFORT ET AL. Matthieu Perrot. Tech. rep. 2011, pp. 2825–2830. URL: <http://scikit-learn.sourceforge.net>. (cit. on p. 9).
- [21] *Randomness Tests Group Website*. <https://randomness-tests.fi.muni.cz/groups>. Accessed on October 16, 2023 (cit. on p. 23).
- [22] *Scipy (v.1.11.3) - Index*. 2023-09. URL: <https://docs.scipy.org/doc/scipy-1.11.3/index.html> (cit. on p. 9).

BIBLIOGRAPHY

- [23] S. H. Strogatz. *Nonlinear Dynamics And Chaos: With Applications To Physics, Biology, Chemistry, And Engineering*. 1st ed. Studies in Nonlinearity. Westview Press, 2001. ISBN: 9780738204536 (cit. on p. 5).
- [24] F. Triefenbach et al. “Acoustic modeling with hierarchical reservoirs”. In: *IEEE Transactions on Audio, Speech and Language Processing* 21.11 (2013), pp. 2439–2450. ISSN: 15587916. DOI: 10.1109/TASL.2013.2280209 (cit. on p. 3).
- [25] T. Waegeman, F. Wyffels, and B. Schrauwen. “Feedback control by online learning an inverse model”. In: *IEEE Transactions on Neural Networks and Learning Systems* 23.10 (2012), pp. 1637–1648. ISSN: 2162237X. DOI: 10.1109/TNNLS.2012.2208655 (cit. on p. 1).
- [26] A. S. Weigend. *Time Series Prediction*. Routledge, 2018-05. ISBN: 9780429492648. DOI: 10.4324/9780429492648 (cit. on p. 11).

A

INTRODUCTORY FIGURES

"Proof for the stability of the first order IIR"

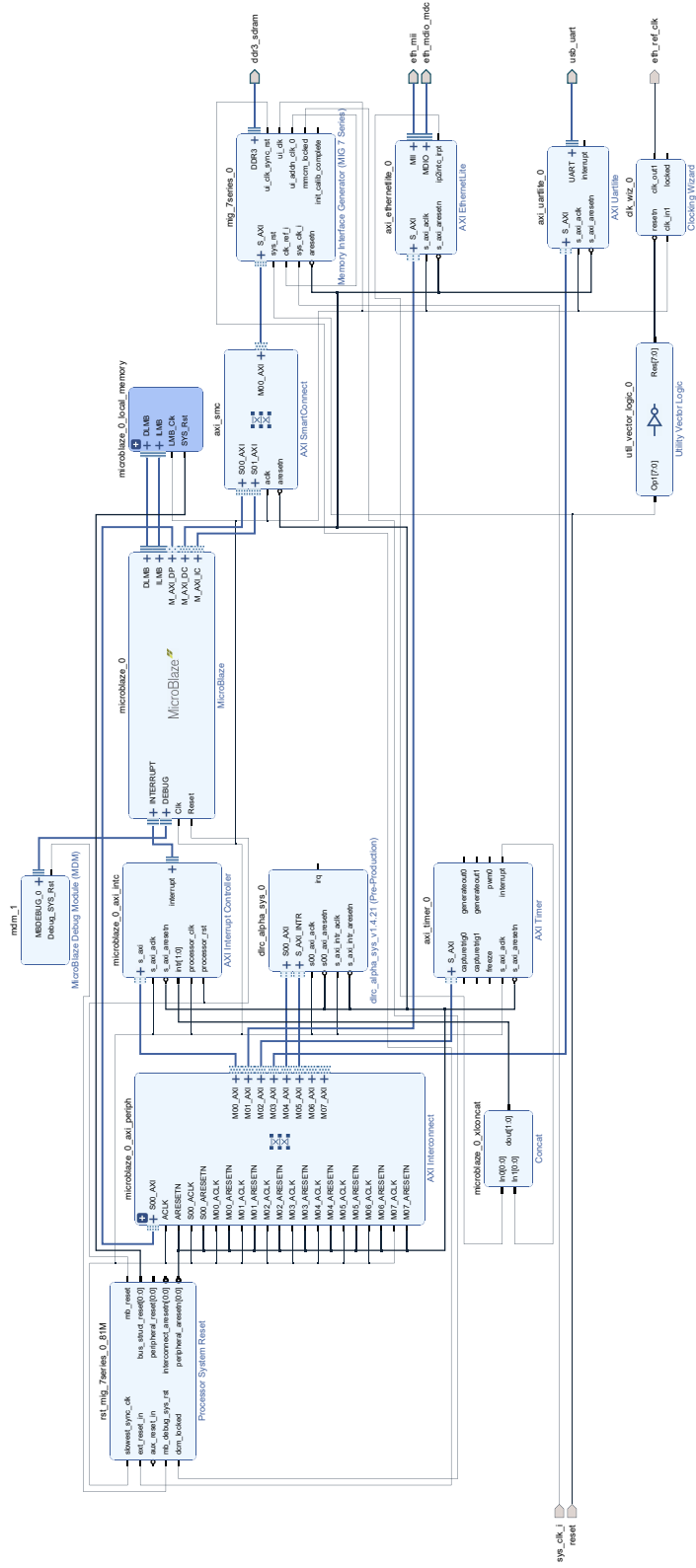


Figure A.1: SoC accelerator block design architecture. Vivado.

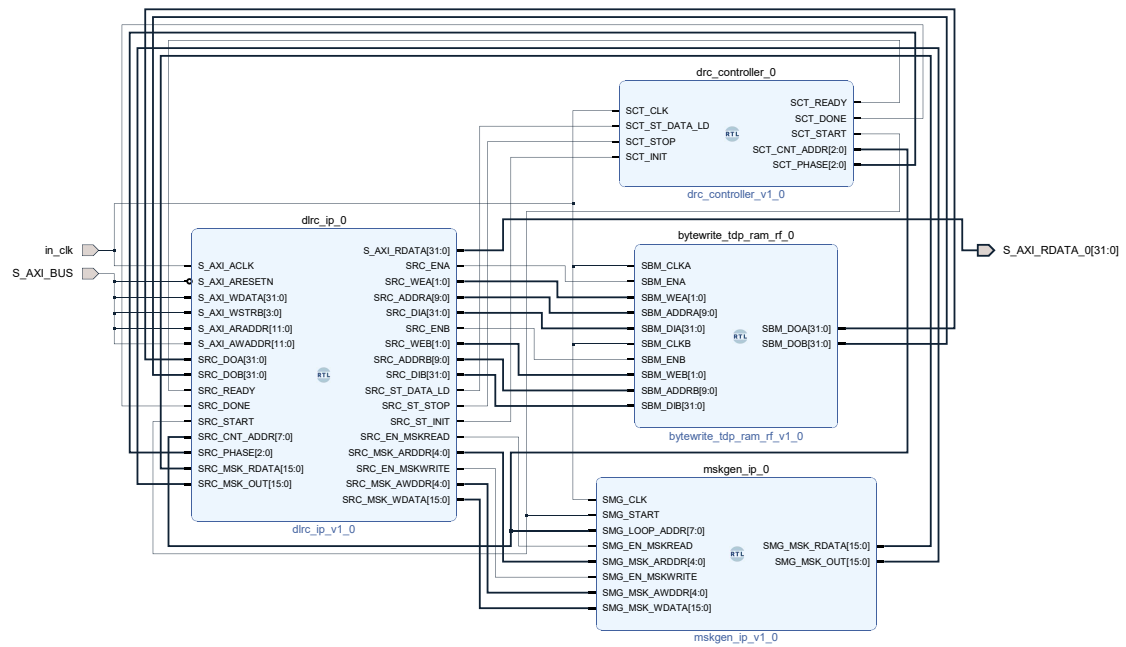


Figure A.2: Delay Line Reservoir Computer IP block design architecture. Vivado.

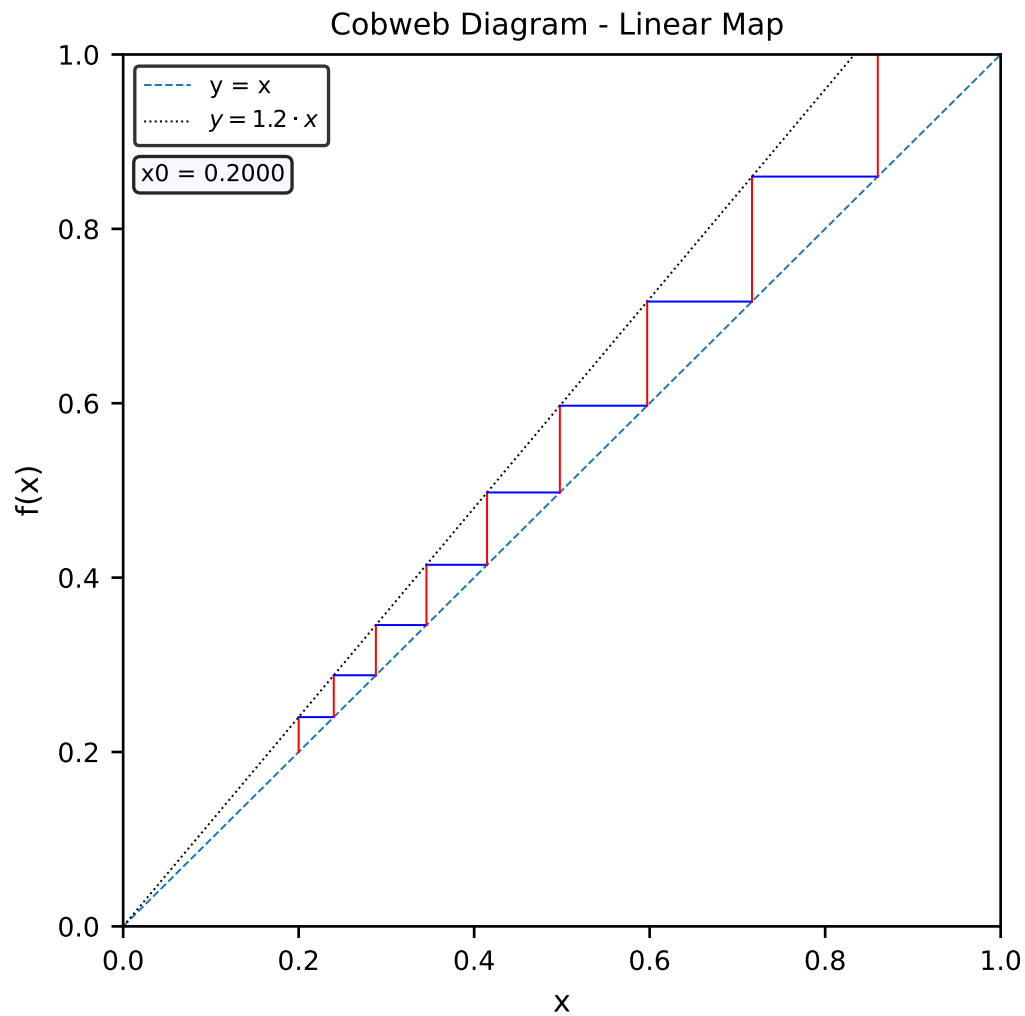


Figure A.3: Cobweb plot of an iterated one-dimensional map using a linear map. Notice how trajectories are dispersed towards infinity, never to return. This is the main reason for attractors to be nonlinear phenomenon.

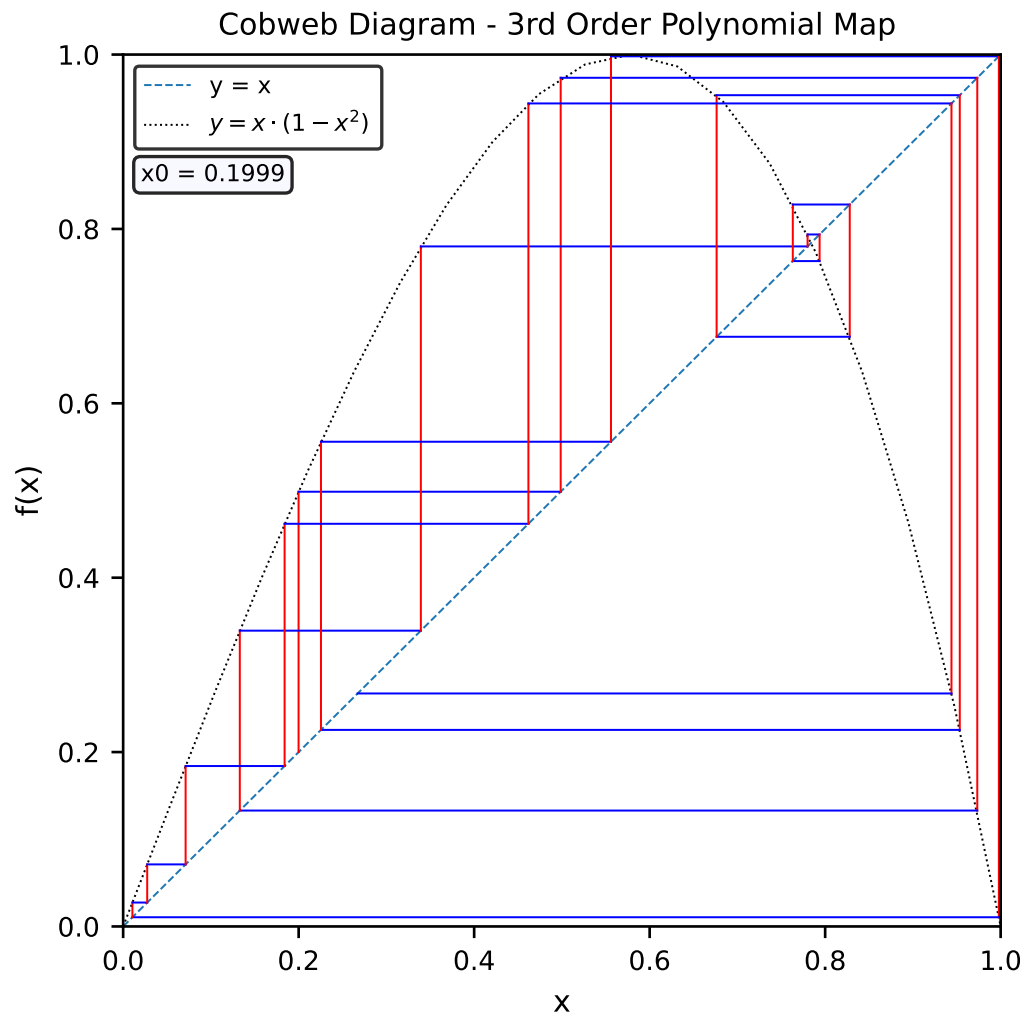


Figure A.4: Cobweb plot of an iterated one-dimensional map using the 3rd order polynomial in (state Eq.). Initial condition = 0.1999

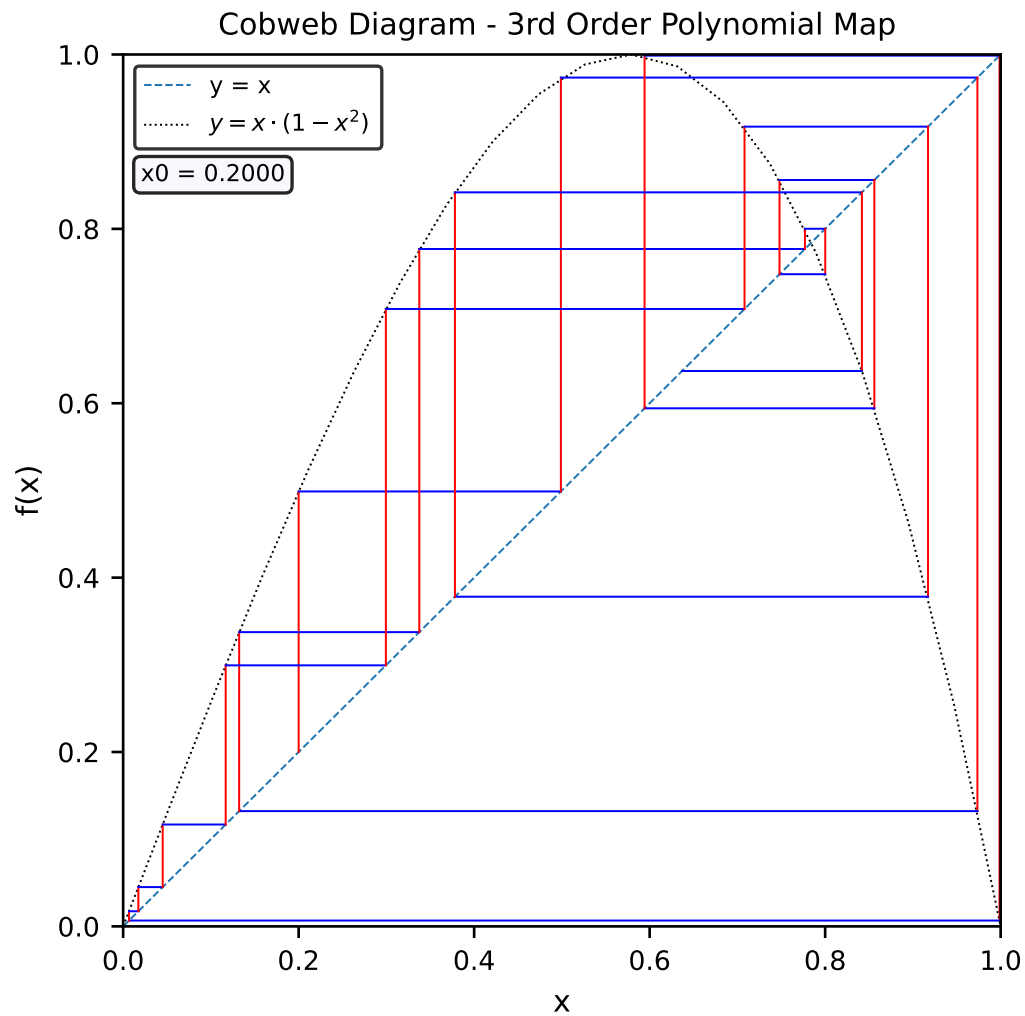


Figure A.5: Cobweb plot of an iterated one-dimensional map using the 3rd order polynomial in (state Eq.). Initial condition = 0.2. Notice how fast, "webs", or trajectories get separated despite starting close to each other. A.4

APPENDIX 2 - HARDWARE SIMULATIONS

```

1  ...
2  /***** Constant Definitions *****/
3  #define READ_WRITE_MUL_FACTOR 0x10
4  #define dlrc_baseaddr 0x44a00000
5
6  /***** Function Definitions *****/
7  /**
8  XStatus DLRC_ALPHA_Reg_SelfTest(void * baseaddr_p)
9  {
10     u32 baseaddr;
11     int write_loop_index;
12     int read_loop_index;
13     int Index;
14
15     baseaddr = (u32) baseaddr_p;
16     typedef struct {
17         volatile uint32_t reserved;
18         volatile uint32_t weights[256];
19         volatile uint32_t mask[32];
20         volatile uint32_t alpha;
21         volatile uint32_t eta;
22         volatile uint32_t gamma;
23         volatile uint32_t init;
24         volatile uint32_t stop;
25     } my_dlrc_t;
26
27     my_dlrc_t* dlrc_ptr = (my_dlrc_t*)dlrc_baseaddr;
28
29

```

Figure B.1: Structure in Embedded C to interface with the microcontroller in our SoC.

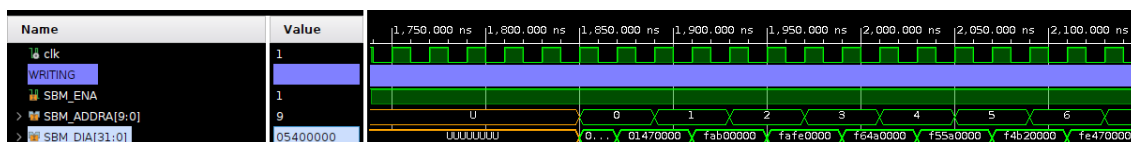


Figure B.2: BRAM's response to the generation of output values. The signal's radix (SBM_DIA and src_current_dv) is Hexadecimal. The top 4 bytes are used for the delay value.

APPENDIX B. APPENDIX 2 - HARDWARE SIMULATIONS

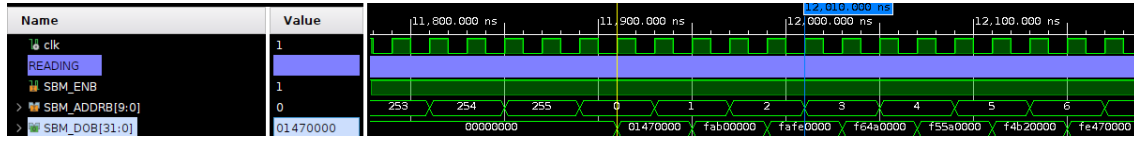


Figure B.3: Retrieval of BRAM values, delay taps and weights. Read during Stage 3 of the Pipeline. Compare values at SBM_DOB with values computed at Stage 6

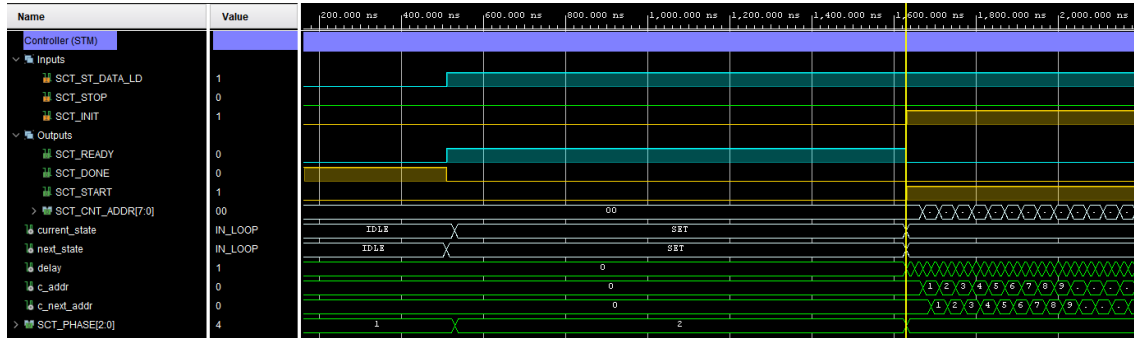


Figure B.4: Timing diagram of the controller's response to inputs until production of addresses commences.

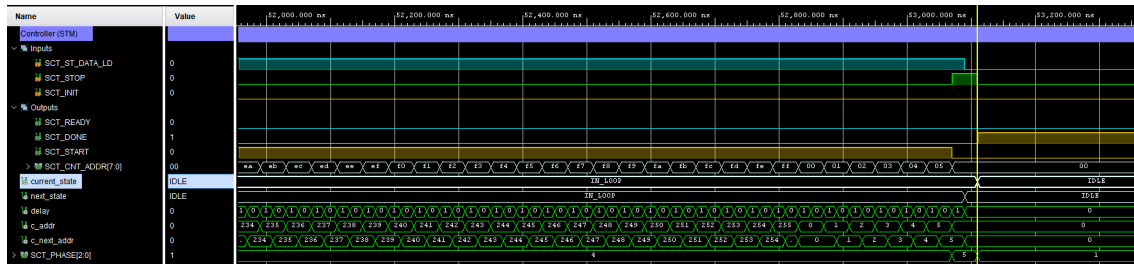


Figure B.5: Timing diagram of the controller upon meeting ending criteria. The controller freezes once no more data remains in the buffer. The microcontroller ensures that there is always at least one input sample in the buffer until every input is processed.

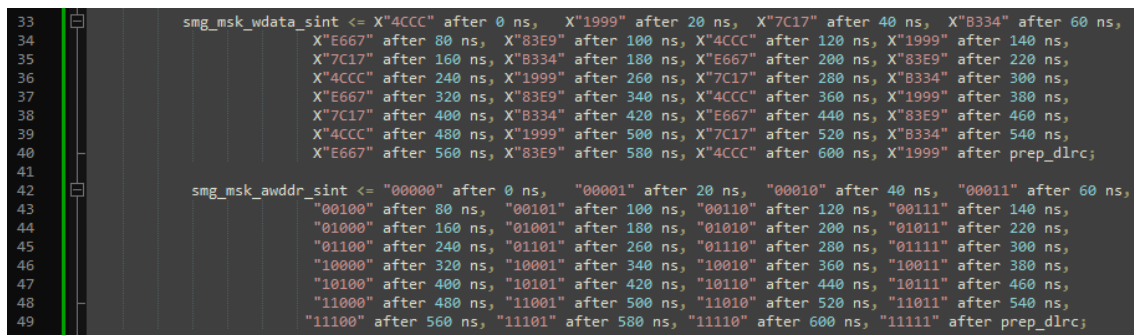
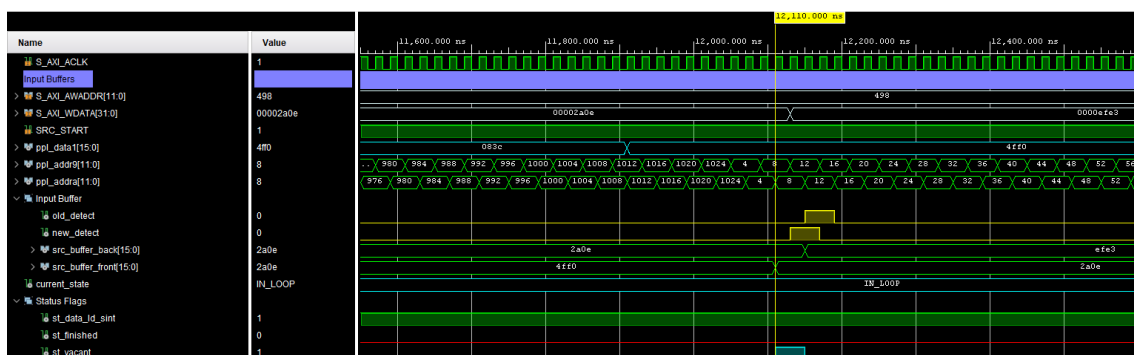
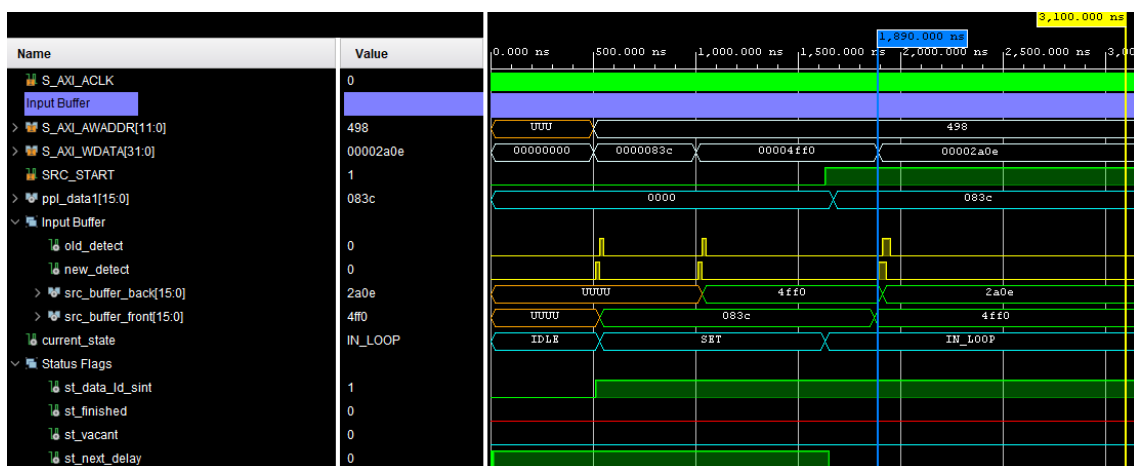
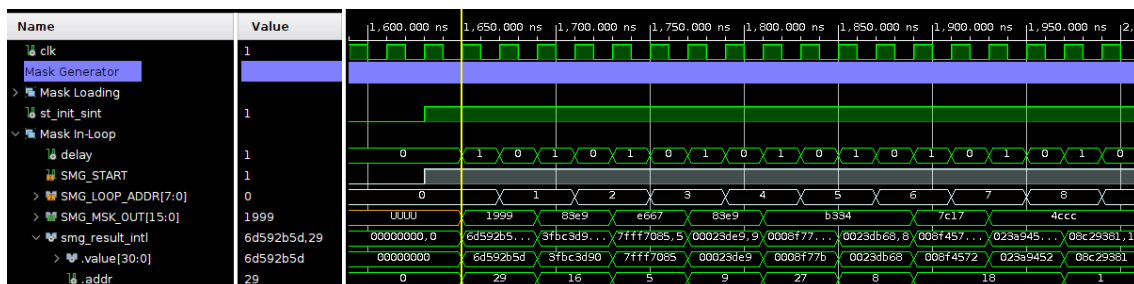
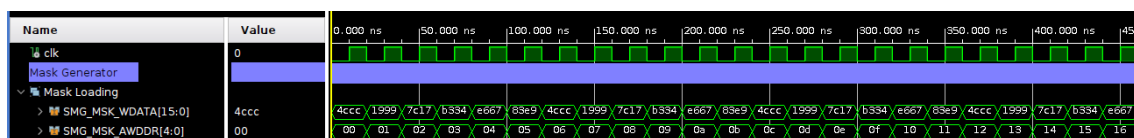


Figure B.6: Testbench code. Loading a six-valued mask to each of the 32 registers in the mask generator. Top are the values being written to the addresses below.



APPENDIX B. APPENDIX 2 - HARDWARE SIMULATIONS

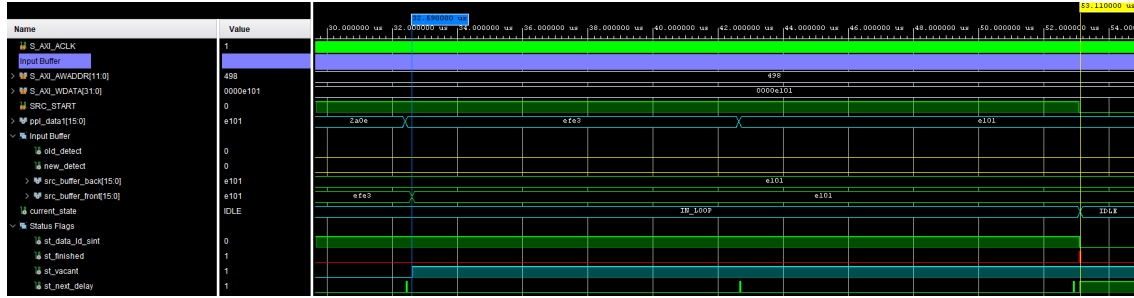


Figure B.11: Timing diagram exhibiting the behaviour of the double buffer implemented at the end.

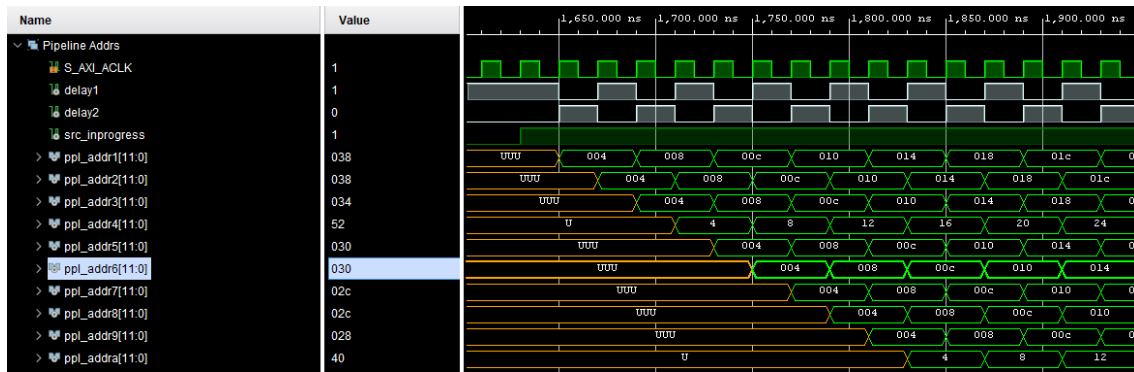


Figure B.12: Pipelined registers in sync with the delay circuit. Reminder: Addresses in hardware have an offset of 4, as the two LSBs are ignored for 32-bit memories. For this reason the increment between addresses is equal to 4

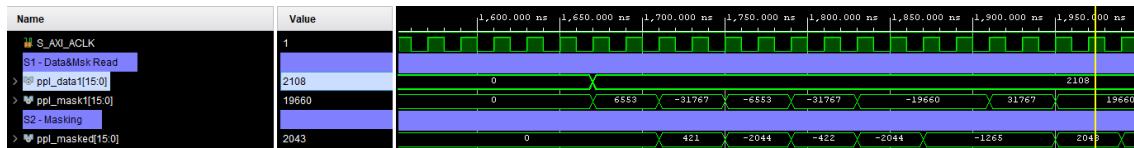


Figure B.13: Stage 2: Masking the input sample.

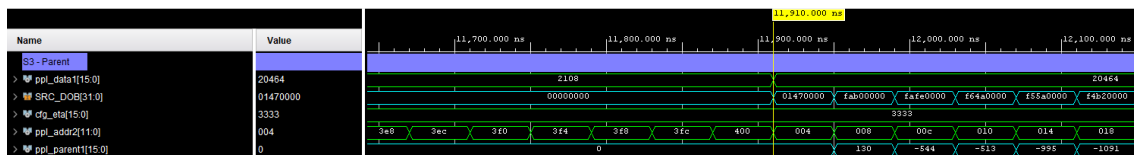


Figure B.14: Stage 3: Reading weights and delay values from the BRAM, using ppl_addr2.

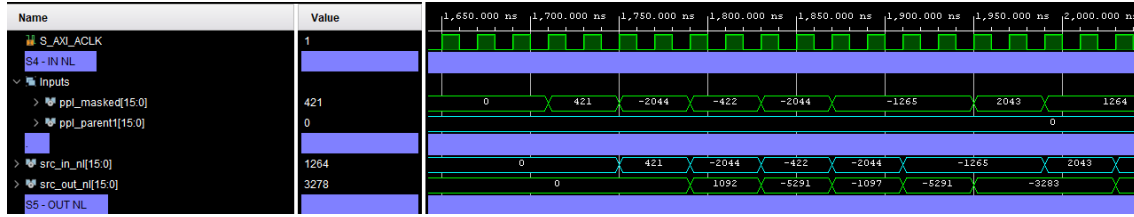


Figure B.15: Stage 4: Combine masked samples with parent values as an input for the NL. For the first sample the BRAM remains empty, hence no parent value can be extracted.

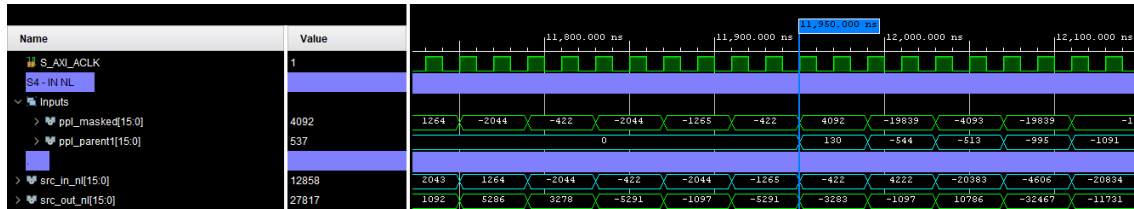


Figure B.16: Stage 4: For the second sample, parent values are now available. The transition between both samples is seen at the blue marker, when the input of the NL is no longer equal to the masked value.

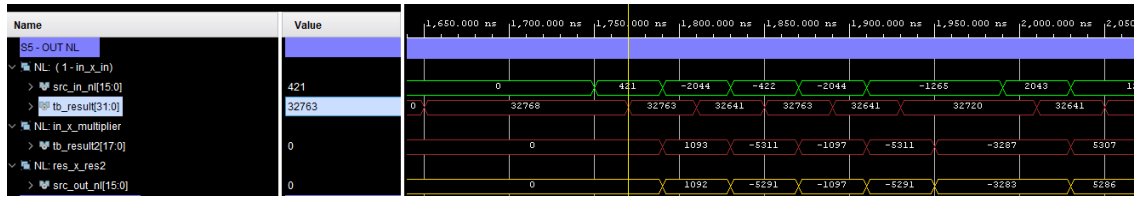


Figure B.17: Non-linearity. Computation in two steps for less overhead.

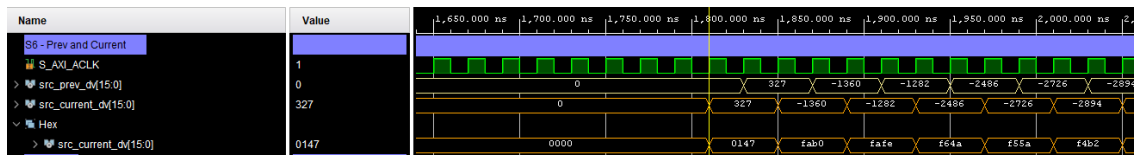


Figure B.18: Value generation from Stage 6 of the Pipeline.

APPENDIX 3 - BENCHMARK RESULTS

C.1 Lorentz Results

C.1.1 3D Plots

C.1.2 Time Series Plots

C.1.3 Regression Plots

C.1.4 9-SA Forecasting

C.1.5 Different initial condition

C.2 Santa Fe Results

C.3 Narma10 Results

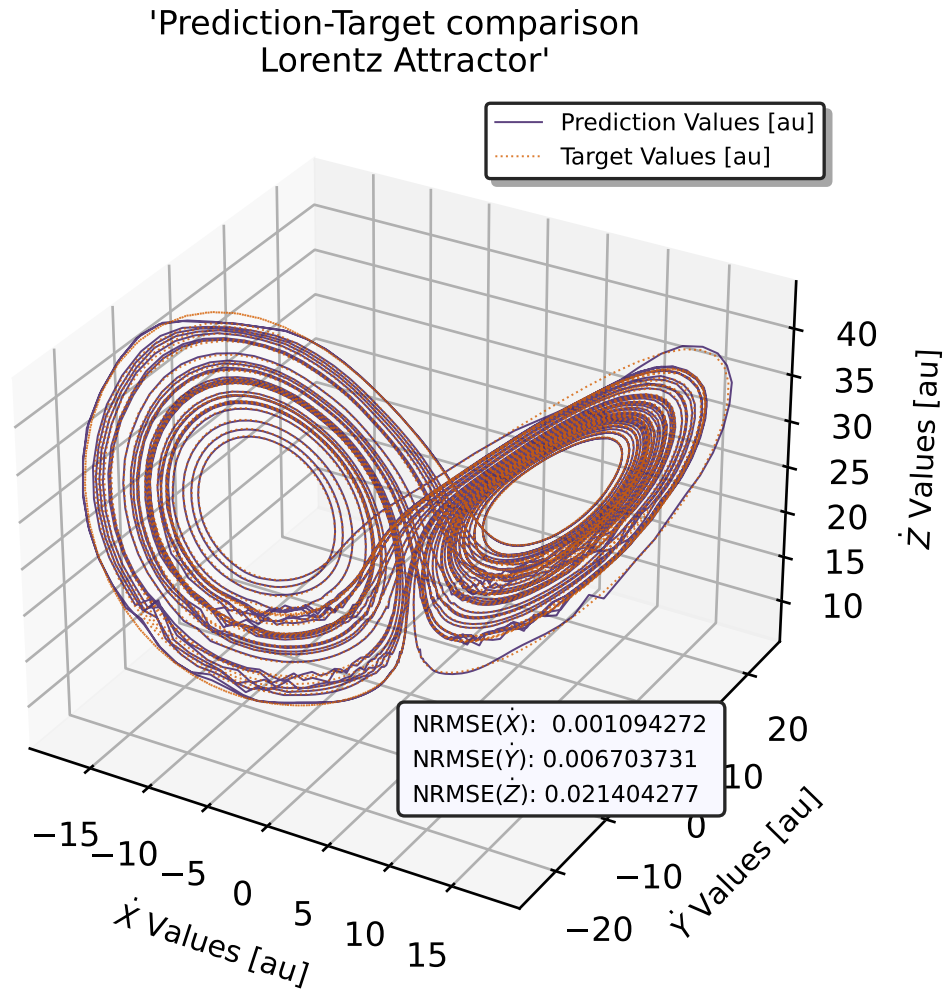


Figure C.1: Comparison between ground truth and predicted attractor. 1-step ahead. Results from using FP numbers in a delay line with 256 VNs.

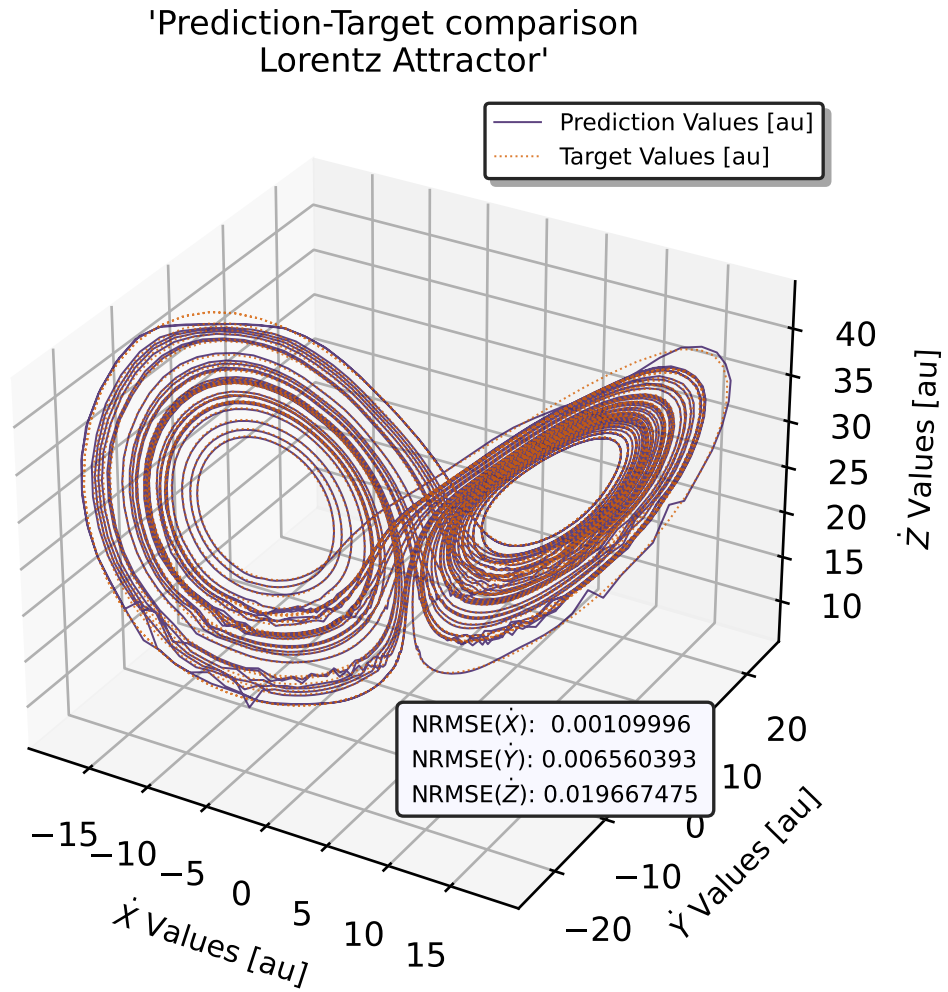


Figure C.2: Comparison between ground truth and predicted attractor. 1-step ahead. Results from using FXP-Int16 numbers in a delay line with 256 VNs.

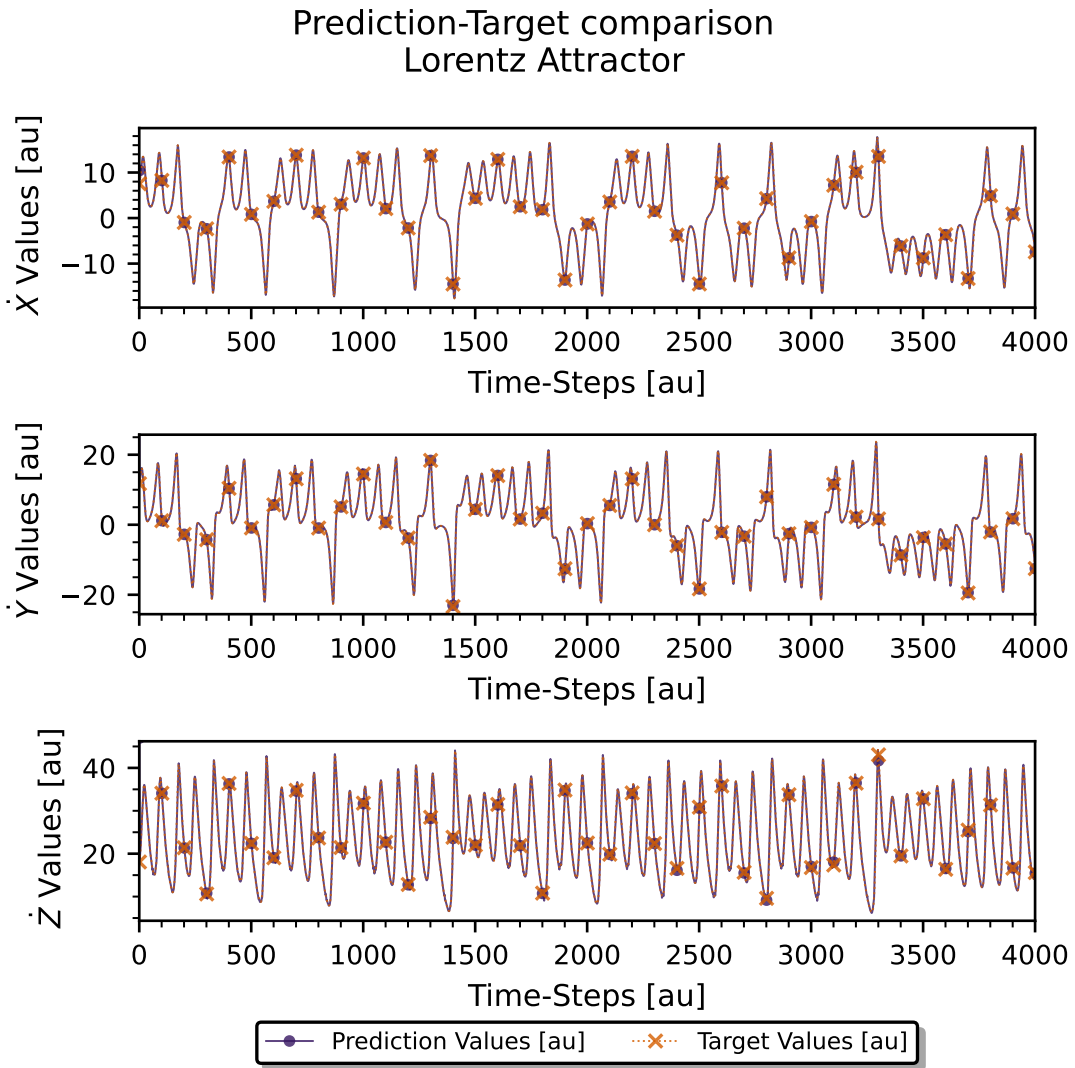


Figure C.3: Comparison between the ground truth dynamics of each isolated variable and respective predictions. 1-step ahead. Results from using FP numbers in a delay line with 256 VNs.

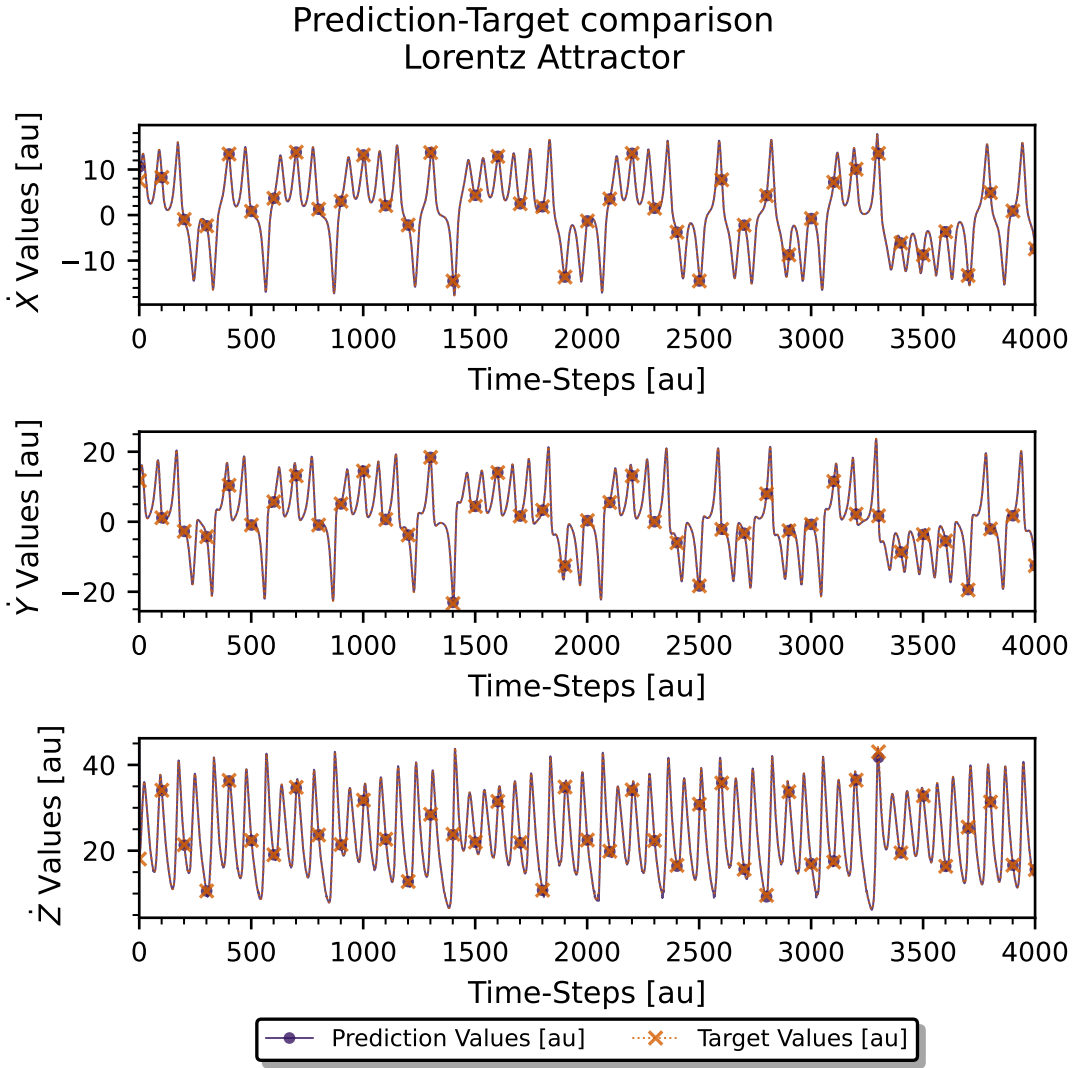


Figure C.4: Comparison between the ground truth dynamics of each isolated variable and respective predictions. 1-step ahead. Results from using FXP-Int16 numbers in a delay line with 256 VNs.

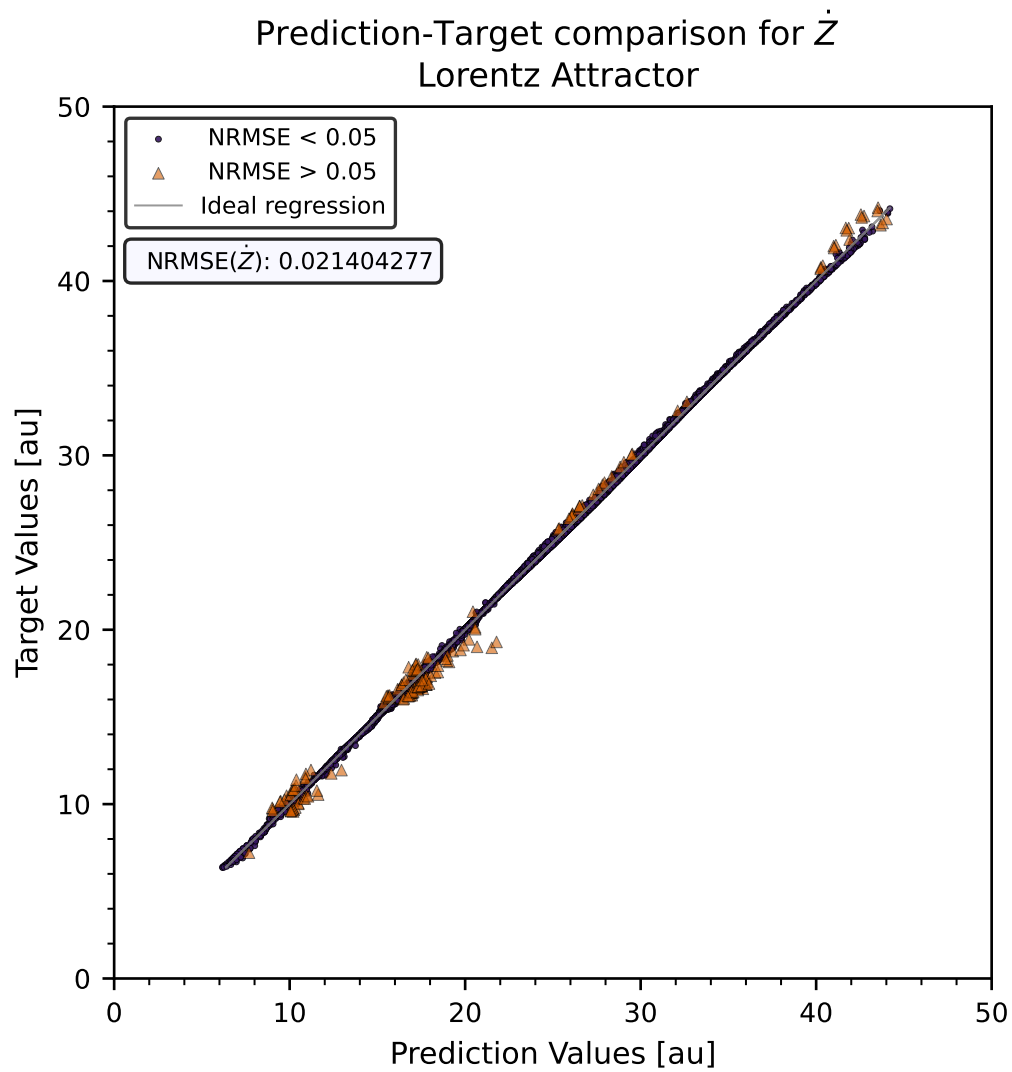


Figure C.5: Linear Regression comparing target values with their respective prediction. Results from using FP numbers in a delay line with 256 VNs. .

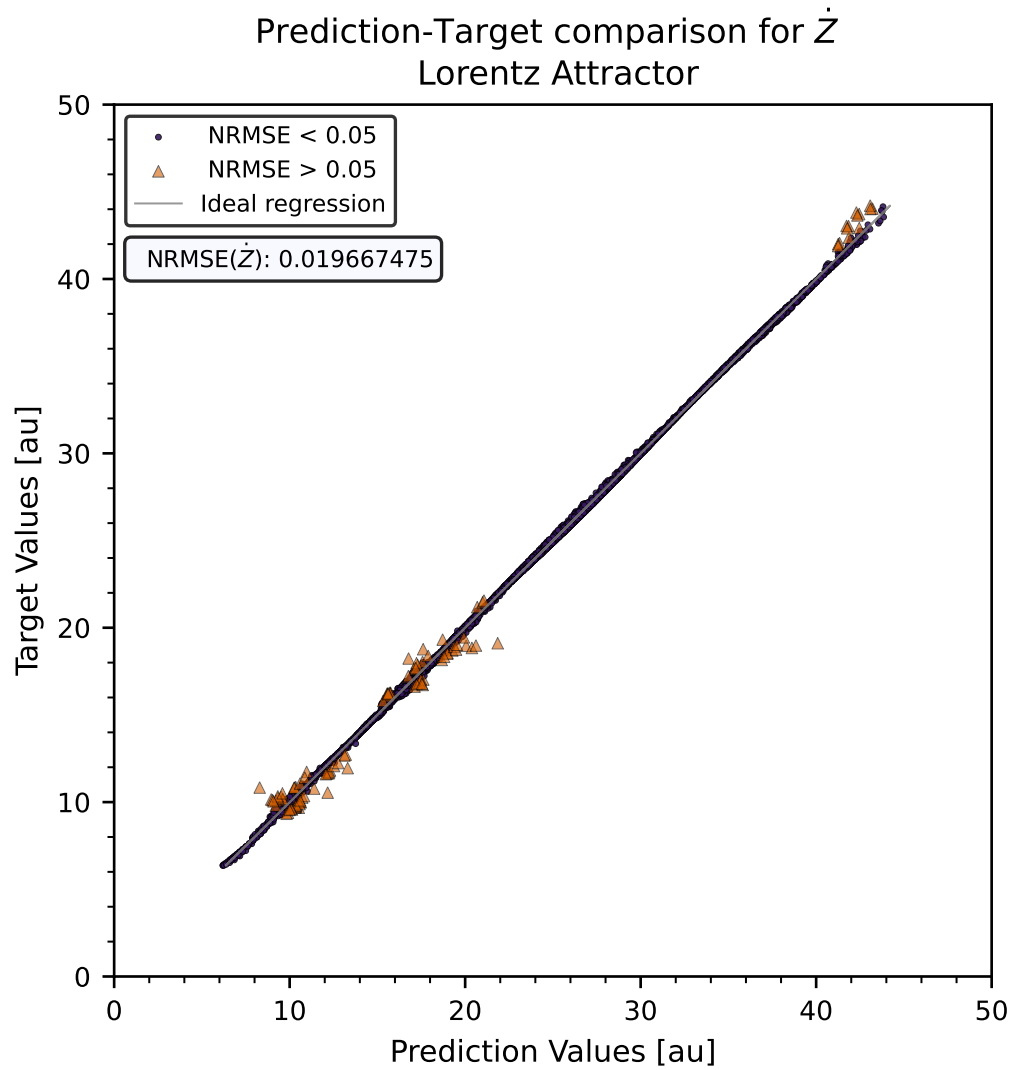


Figure C.6: Linear Regression comparing target values with their respective prediction. Results from using FXP-Int16 numbers in a delay line with 256 VNs. .

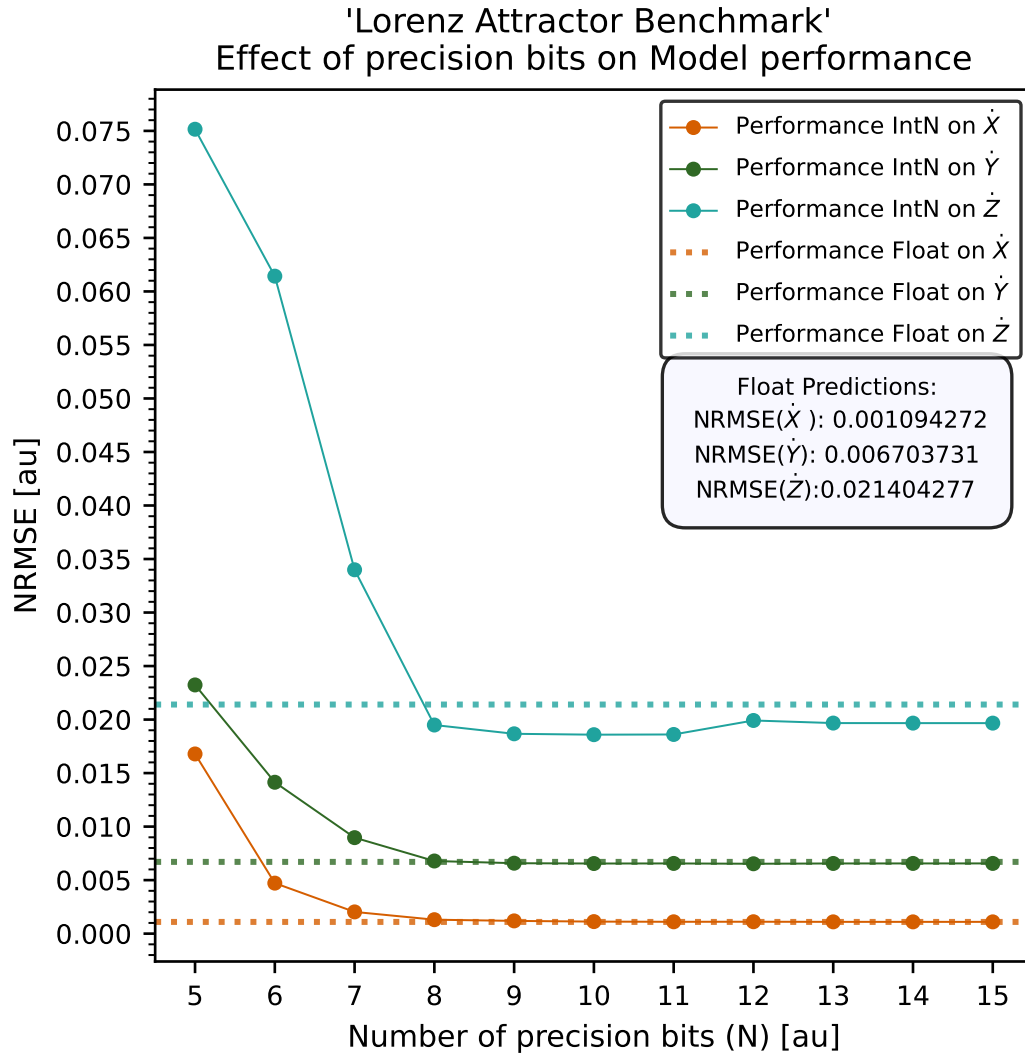


Figure C.7: Effect of number precision in the performance of the DLRC. Horizontal lines correspond to best results from other experimental setups.

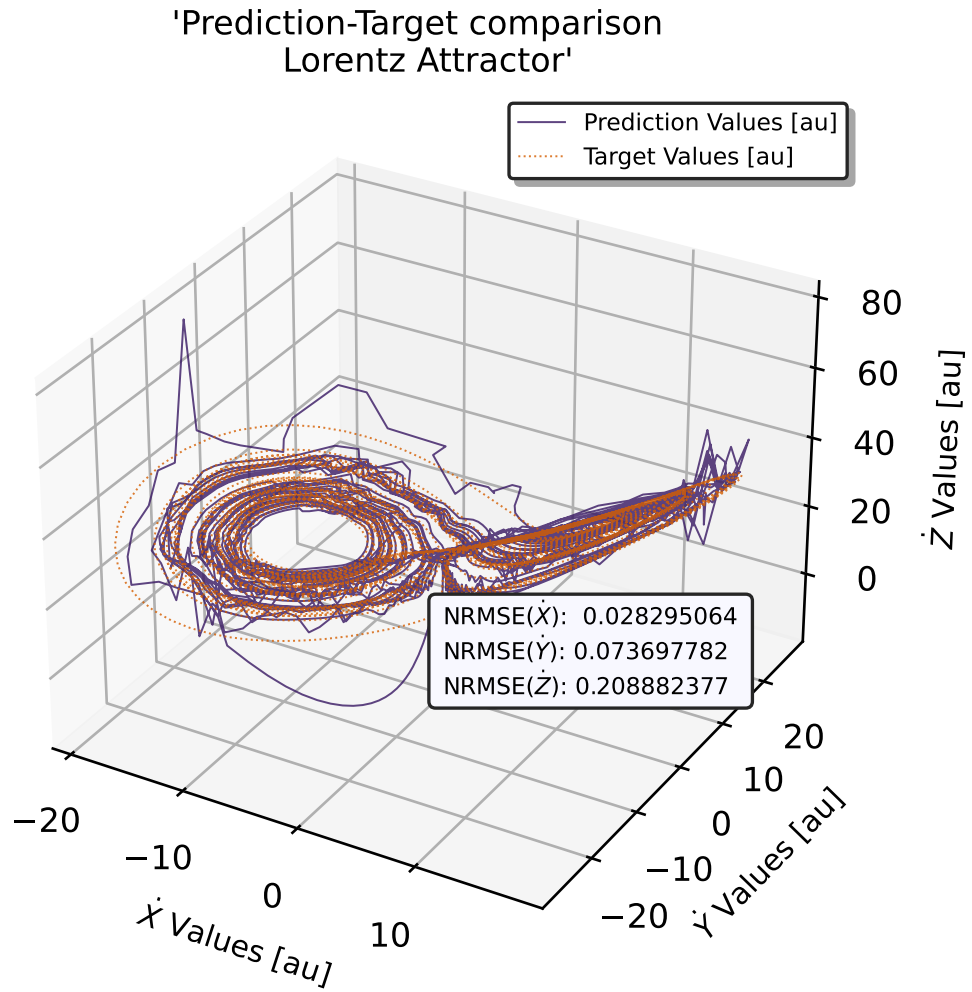


Figure C.8: 9-steps ahead. Comparison between ground truth and predicted attractor. Results from using FP numbers in a delay line with 256 VNs.

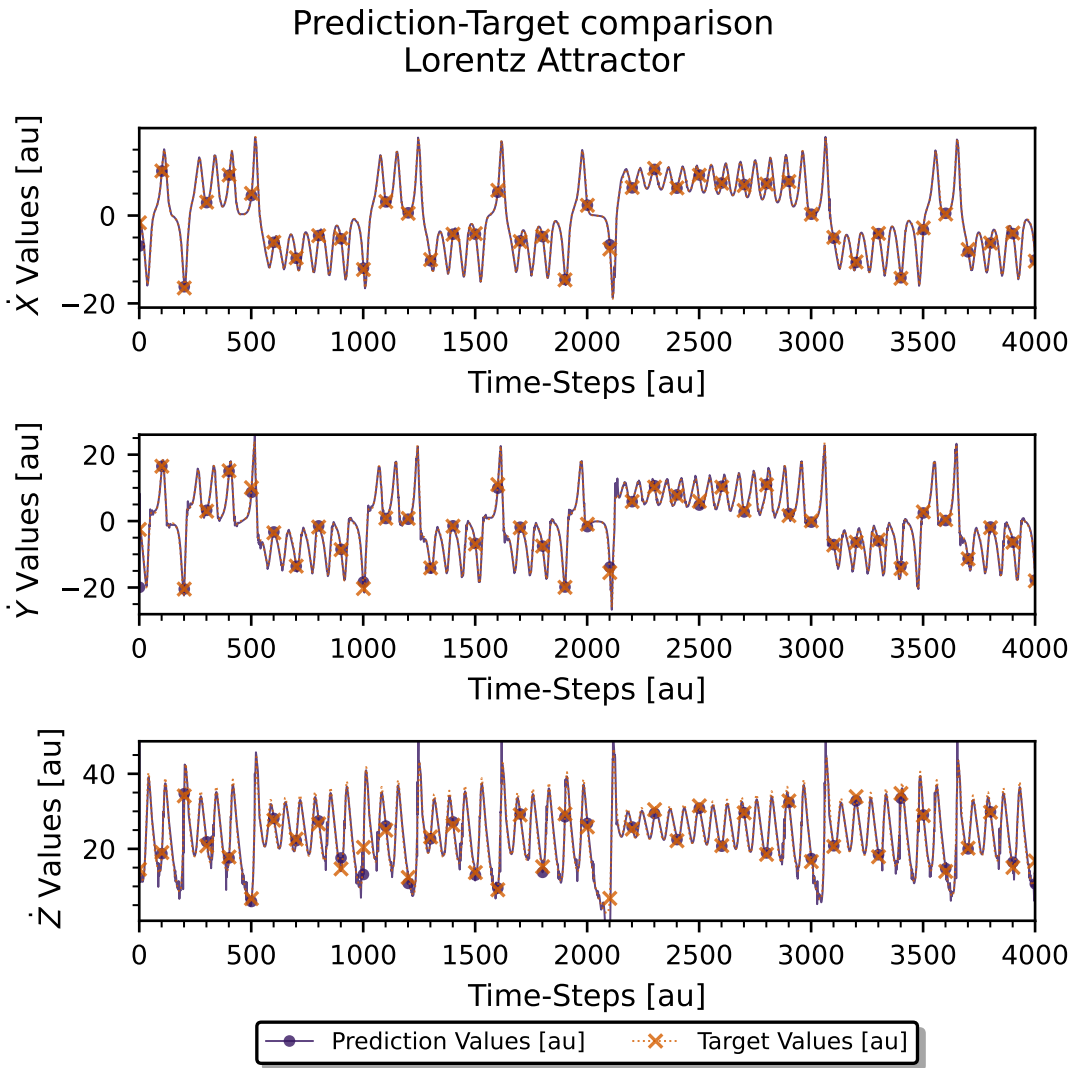


Figure C.9: 9-steps ahead. Comparison between the ground truth dynamics of each isolated variable and respective predictions. Results from using FP numbers in a delay line with 256 VNs.

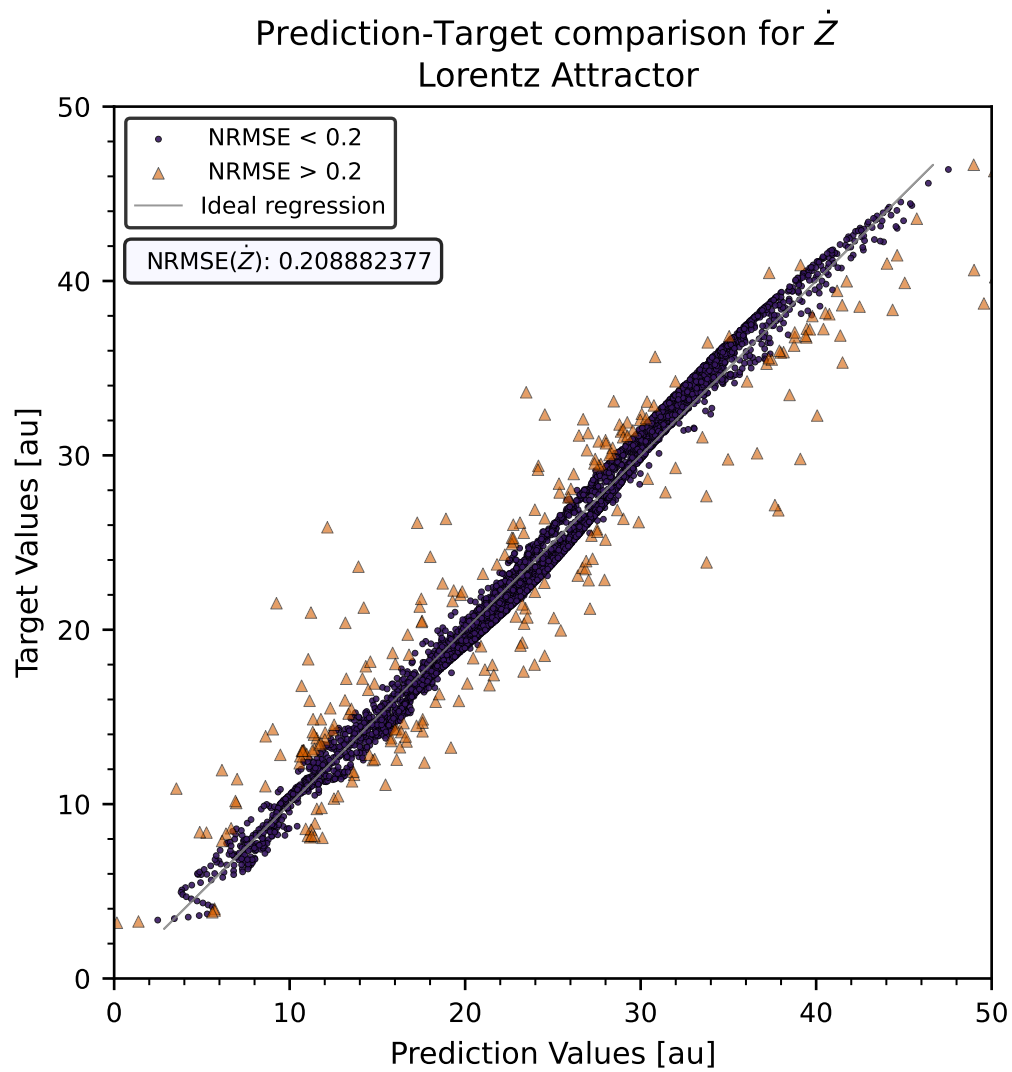


Figure C.10: 9-steps ahead. Linear Regression comparing target values with their respective prediction. Results from using FP numbers in a delay line with 256 VNs..

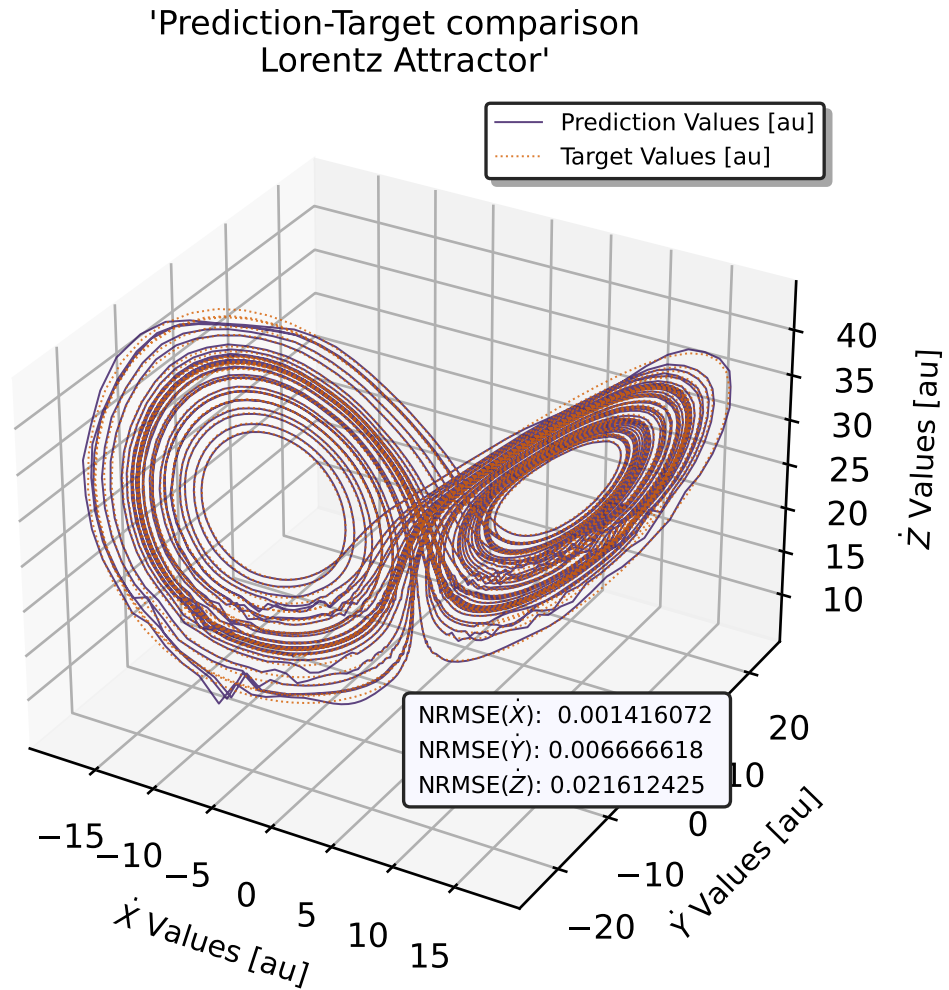


Figure C.11: Different Initial condition. Comparison between ground truth and predicted attractor. 1-step ahead. Results from using FP numbers in a delay line with 256 VNs.

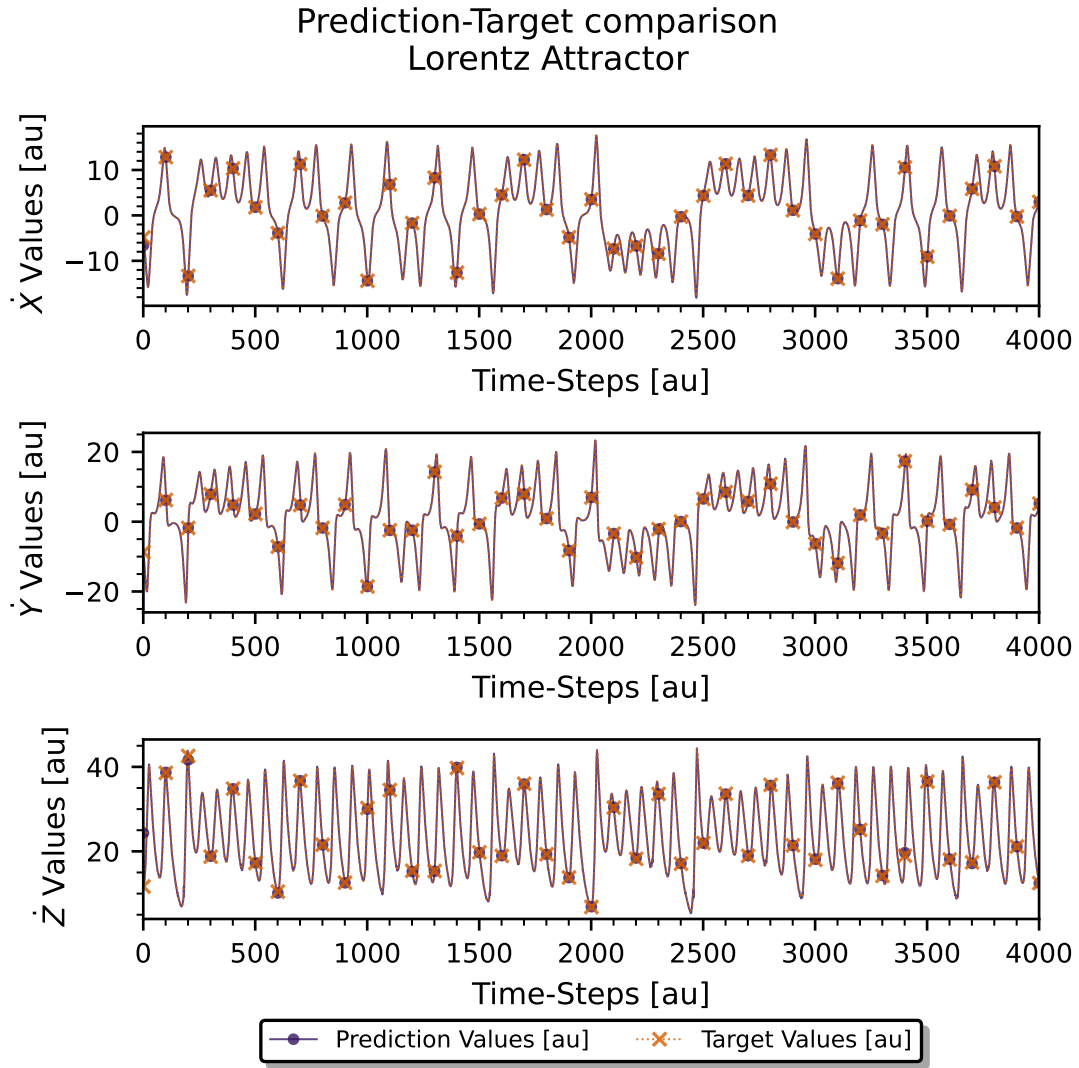


Figure C.12: Different Initial condition. Comparison between the ground truth dynamics of each isolated variable and respective predictions. 1-step ahead. Results from using FP numbers in a delay line with 256 VNs.

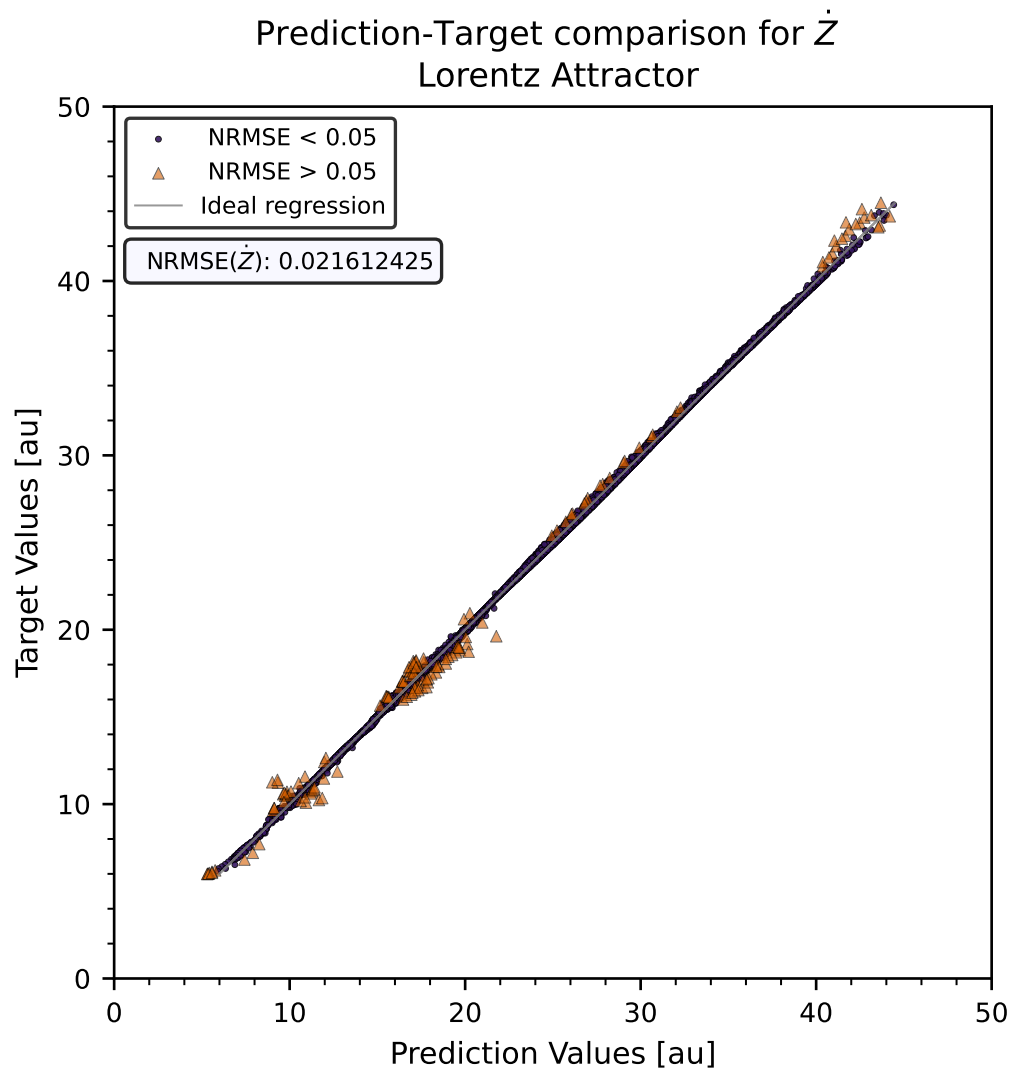


Figure C.13: Linear Regression comparing target values with their respective prediction. Results from using FP numbers in a delay line with 256 VNs.

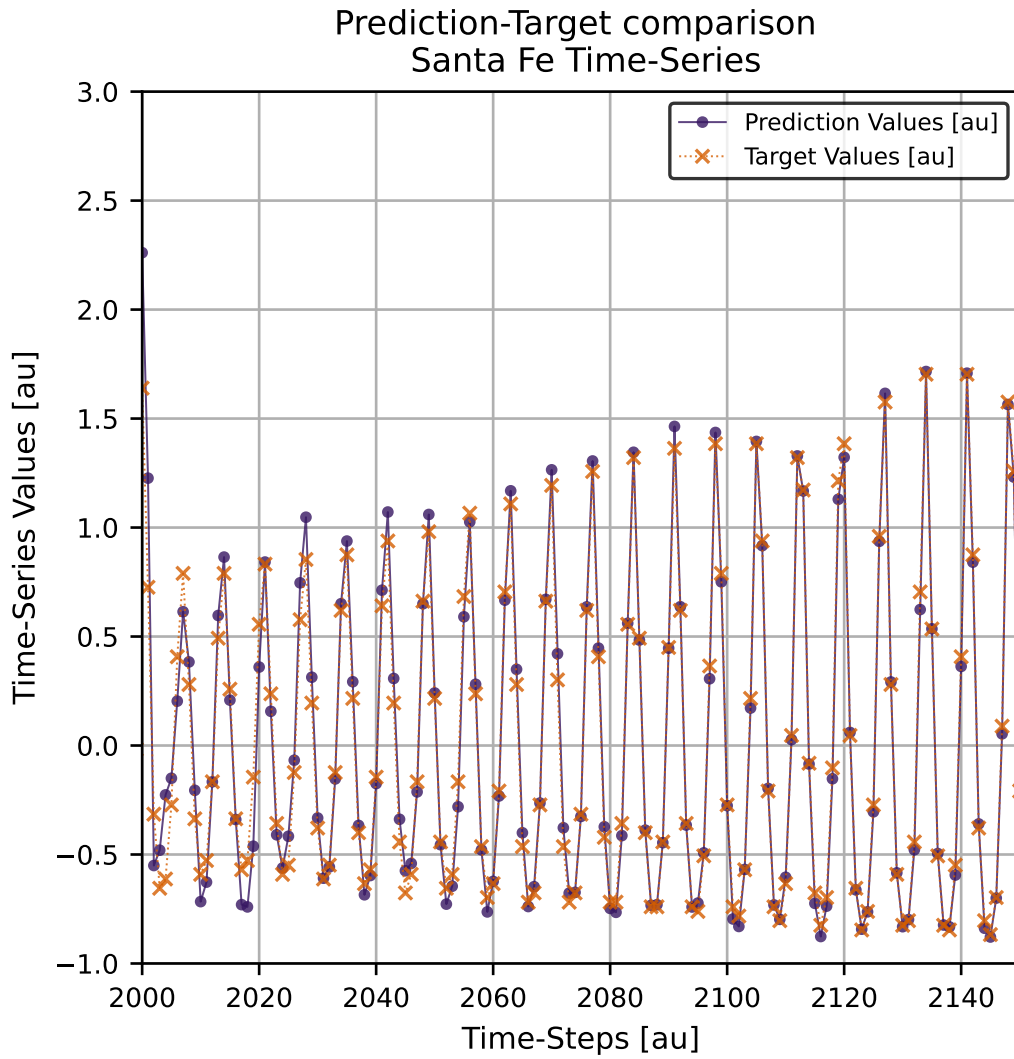


Figure C.14: Comparison between ground truth and predicted time series. Results from using FP numbers in a delay line with 256 VNs.

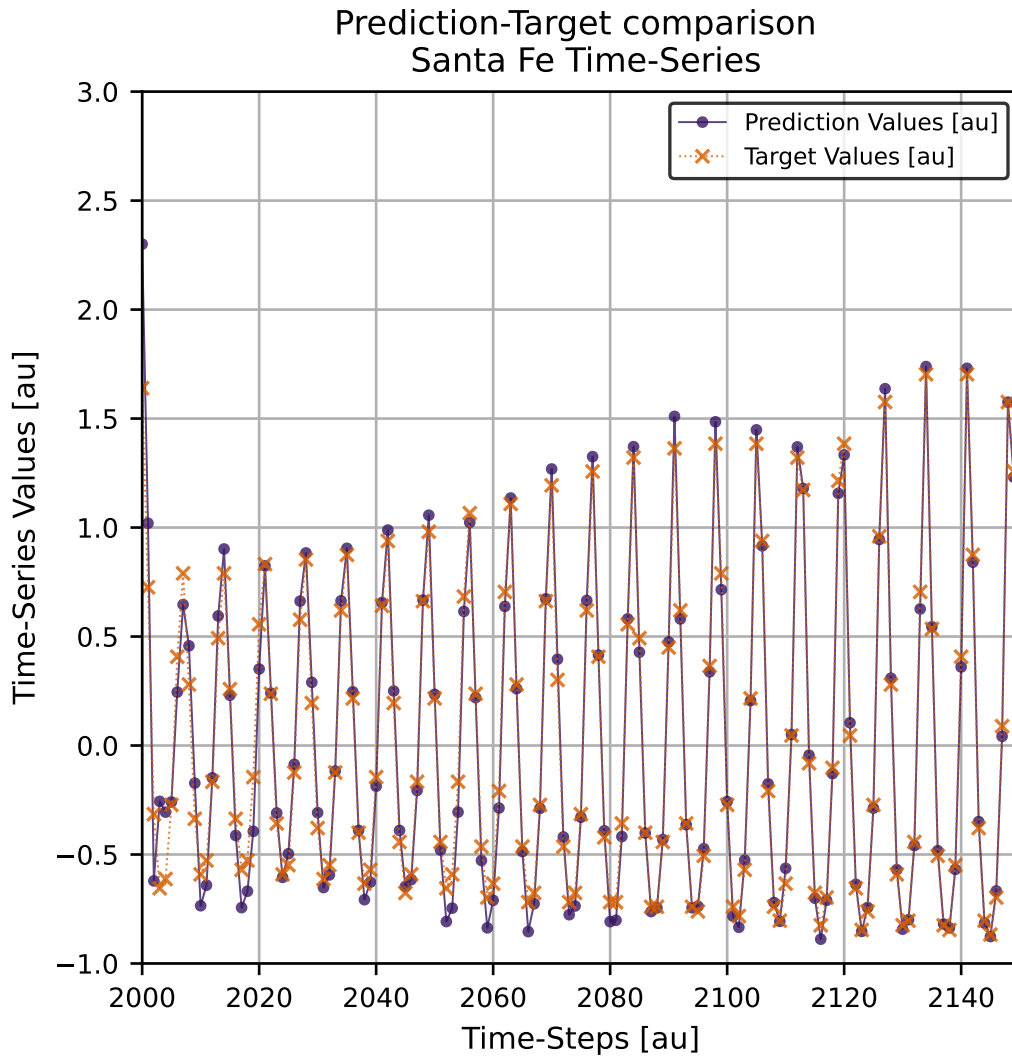


Figure C.15: Comparison between ground truth and predicted time series. Results from using FXP-Int16 numbers in a delay line with 256 VNs.

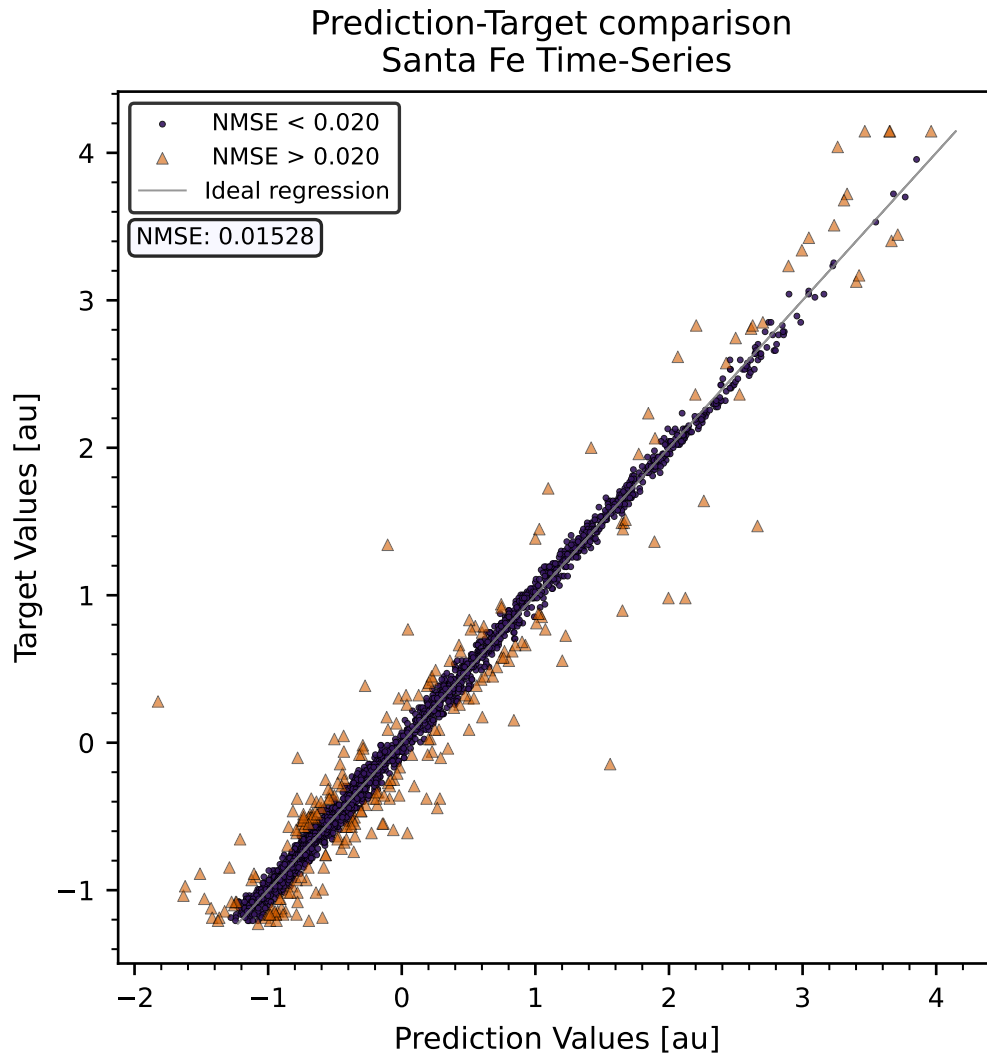


Figure C.16: Linear Regression comparing target values with their respective prediction. 1-step ahead. Results from using FP numbers in a delay line with 256 VNs.

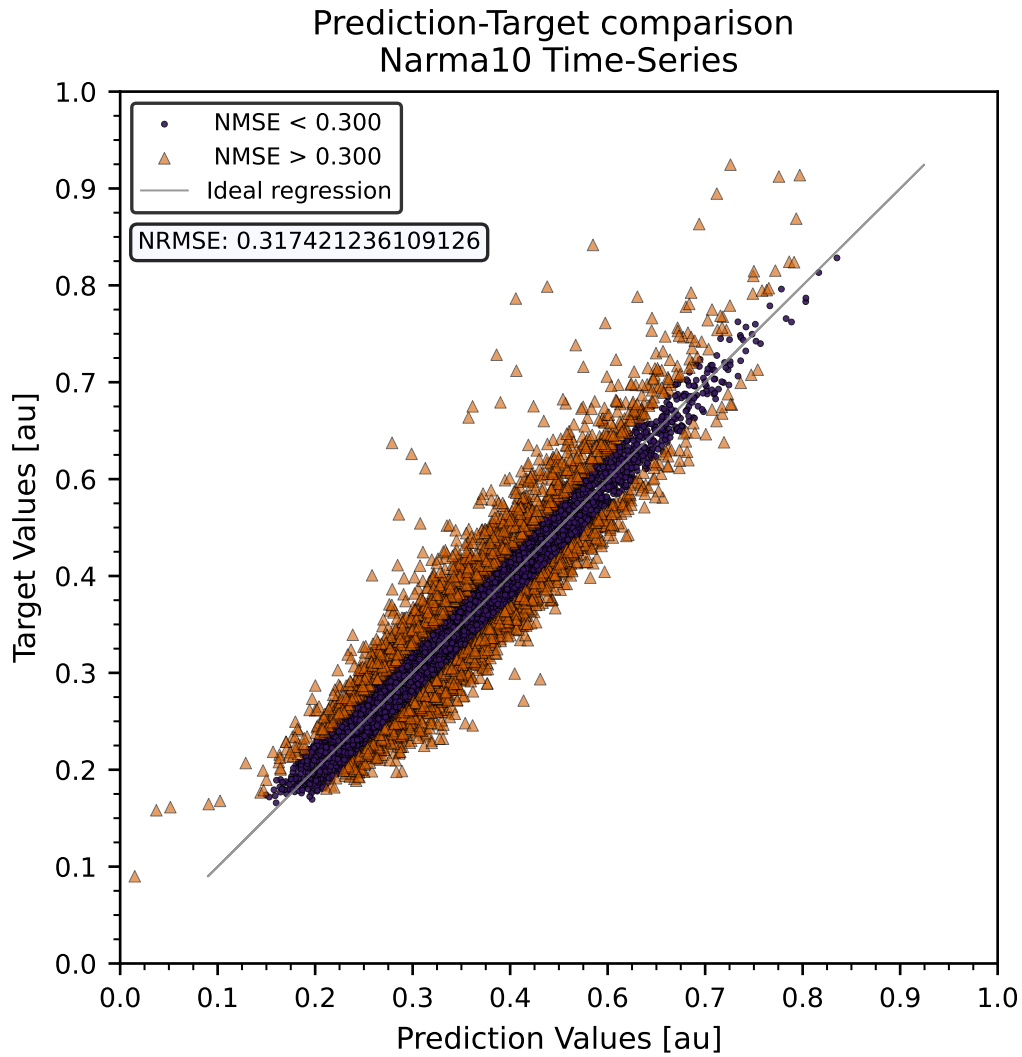


Figure C.17: Linear Regression comparing target values with their respective prediction. 1-step ahead. Results from using FXP numbers in a delay line with 256 VNs.

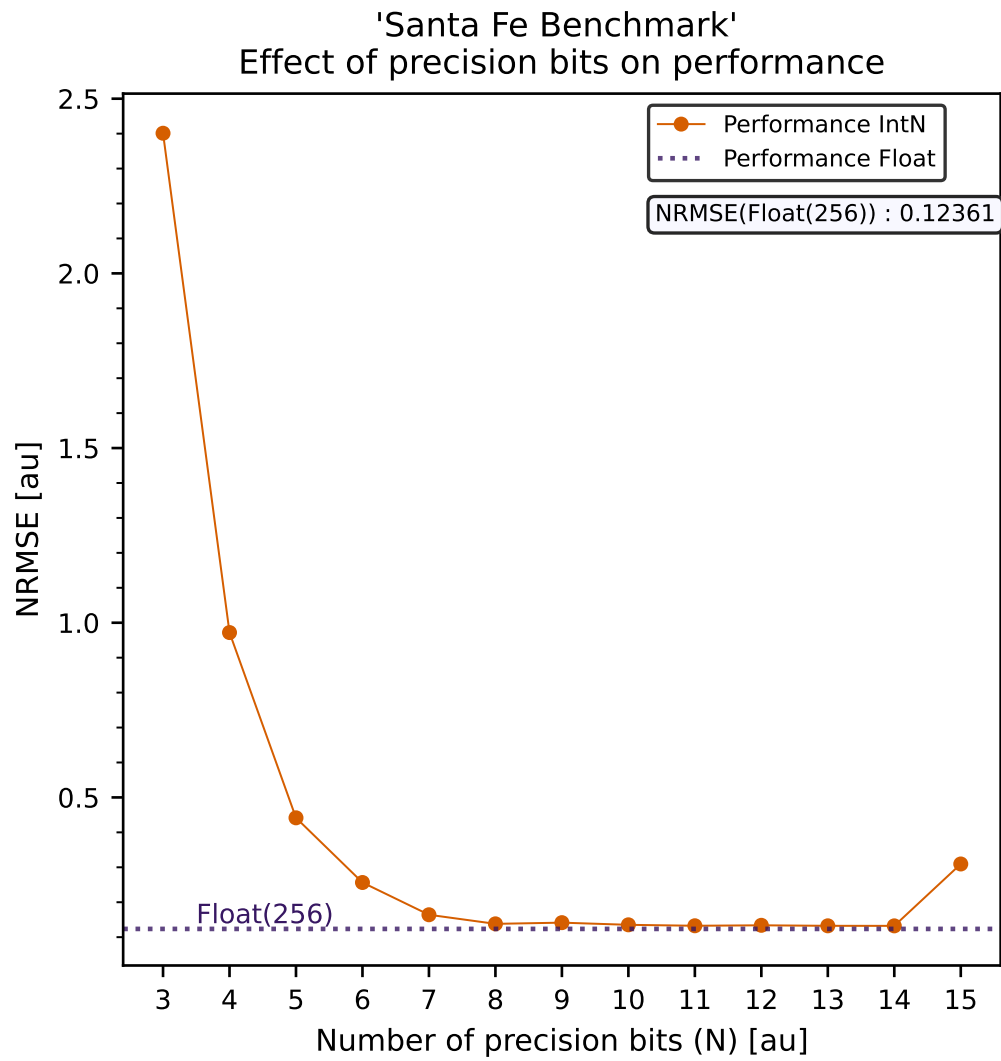


Figure C.18: Effects of number precision in the performance of the DLRC. Horizontal lines correspond to the best results from other experimental setups.

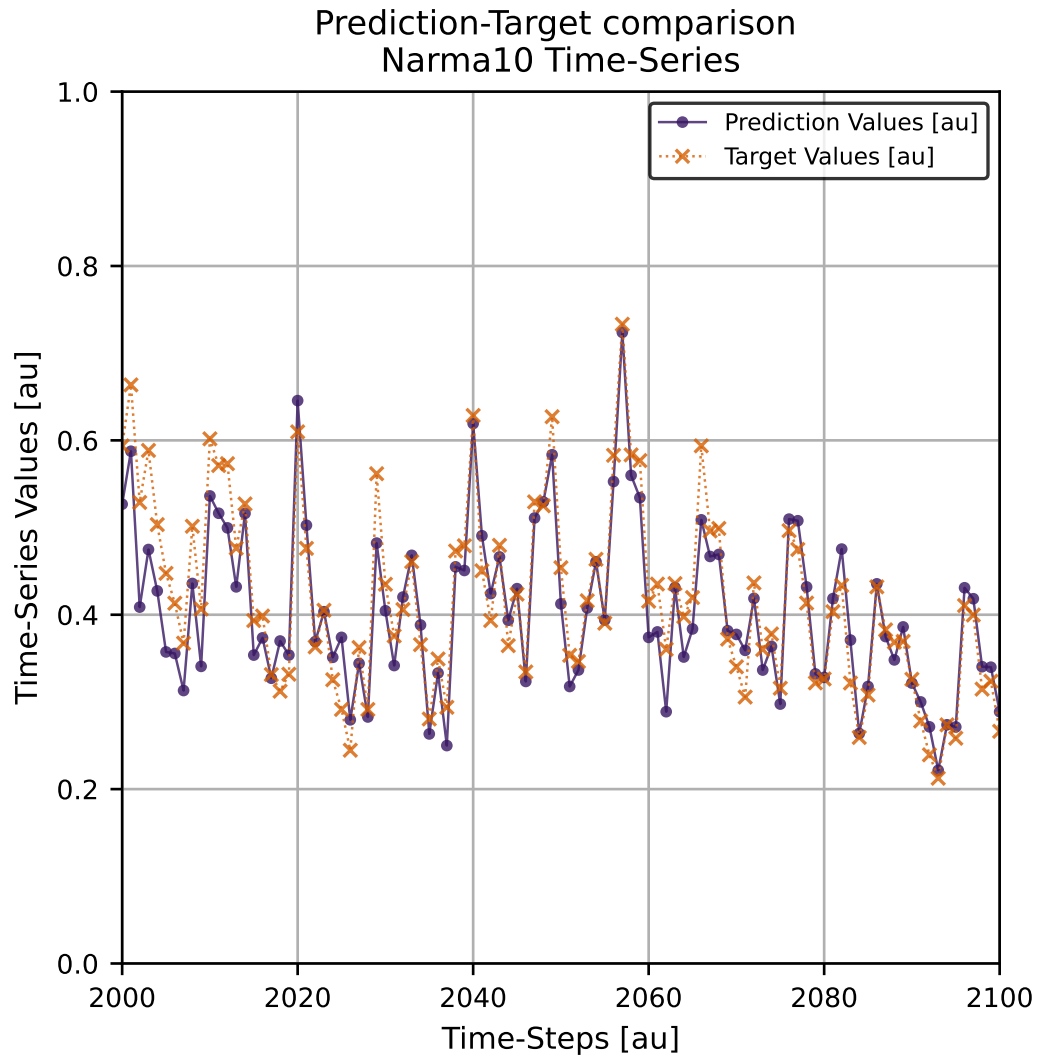


Figure C.19: Effects of number precision in the performance of the DLRC. Horizontal lines correspond to the best results from other experimental setups.

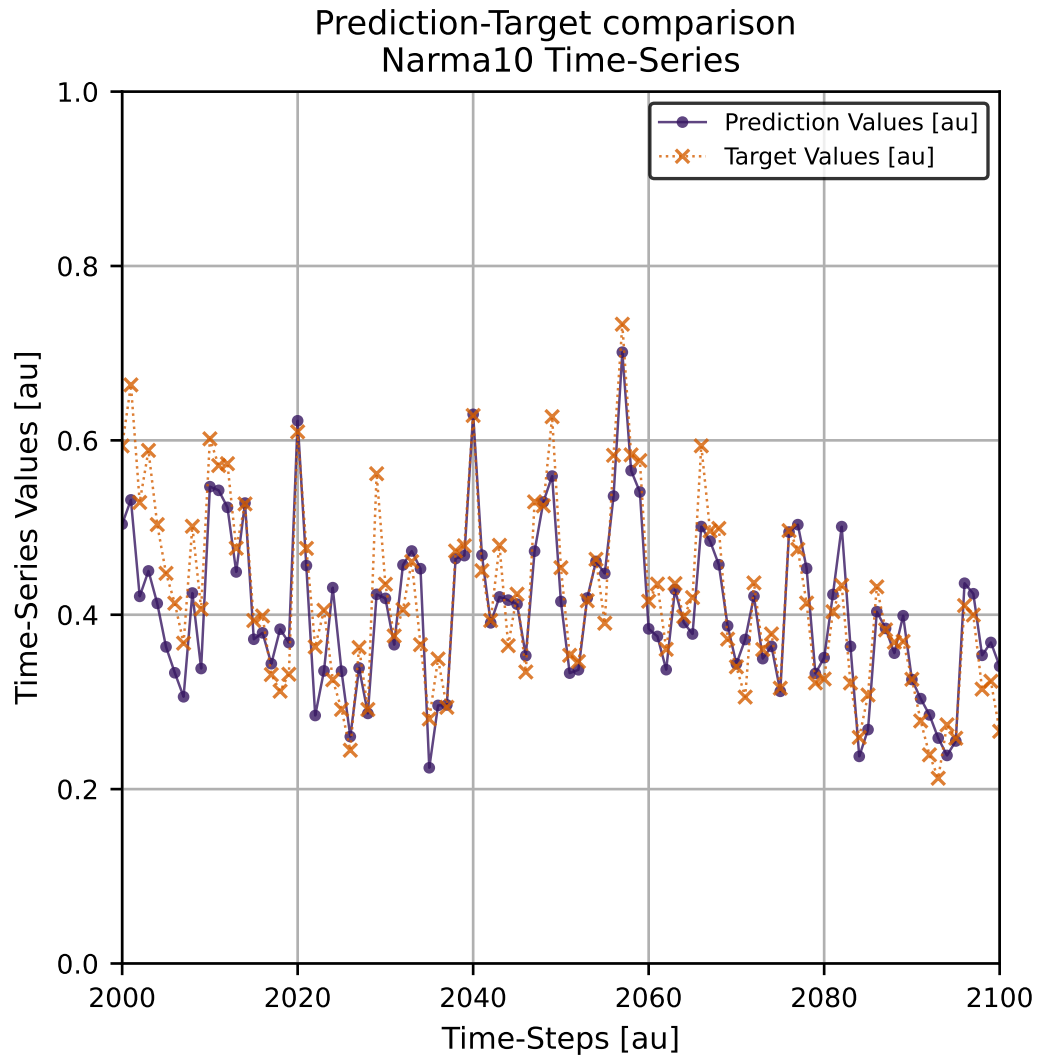


Figure C.20: Effects of number precision in the performance of the DLRC. Horizontal lines correspond to the best results from other experimental setups.

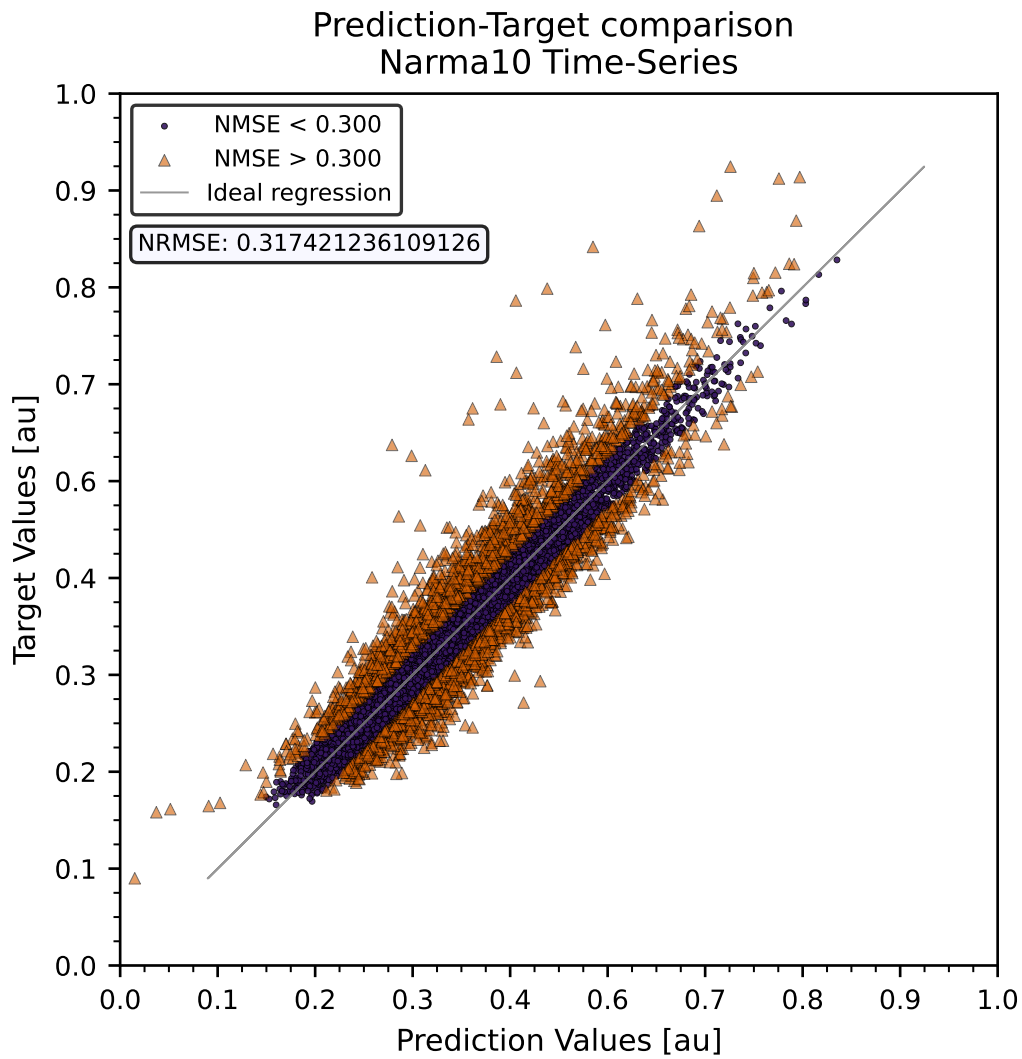


Figure C.21: Comparison between the Narma10 and the mimicked systems obtained from the DLRC. Results from using FP numbers in a delay line with 256 VNs.

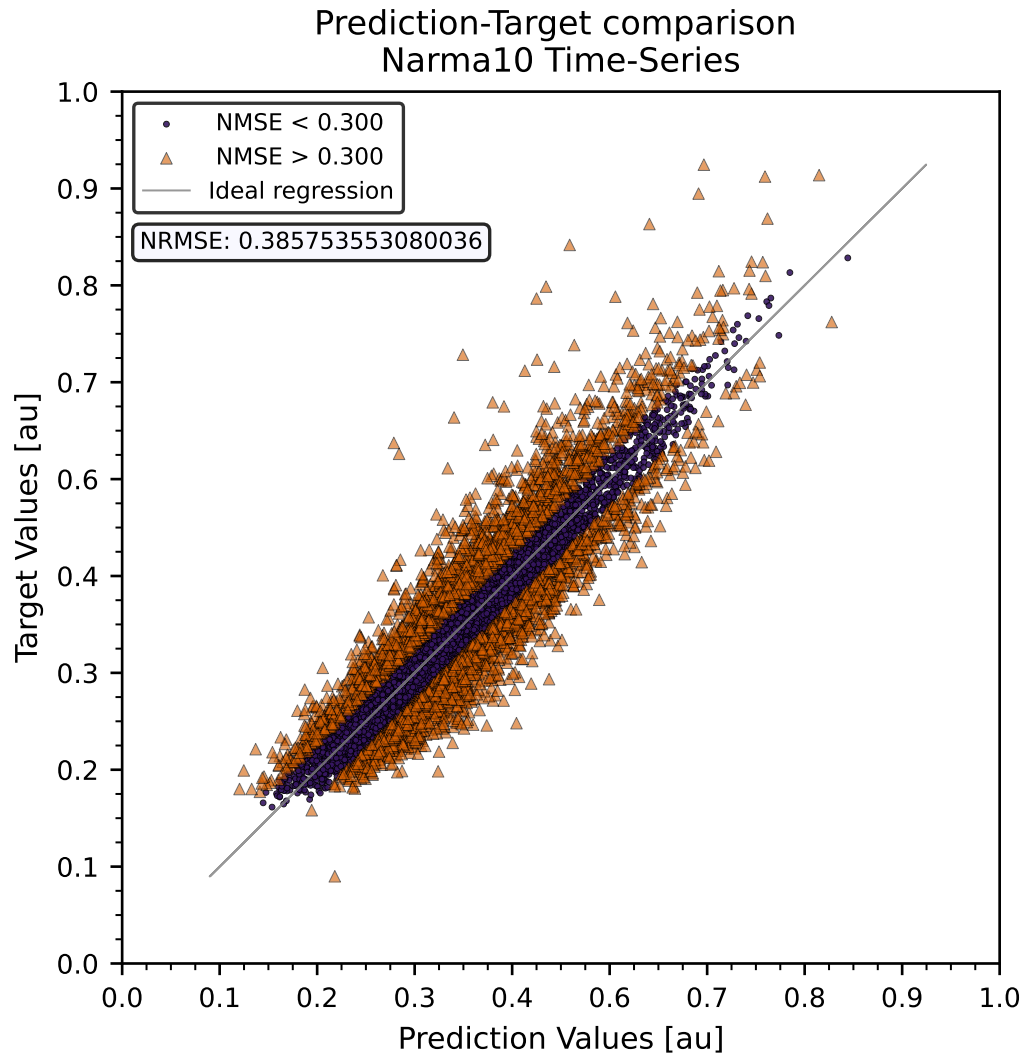


Figure C.22: Comparison between the Narma10 and the mimicked systems obtained from the DLRC. Results from using FXP-Int16 numbers in a delay line with 256 VNs.

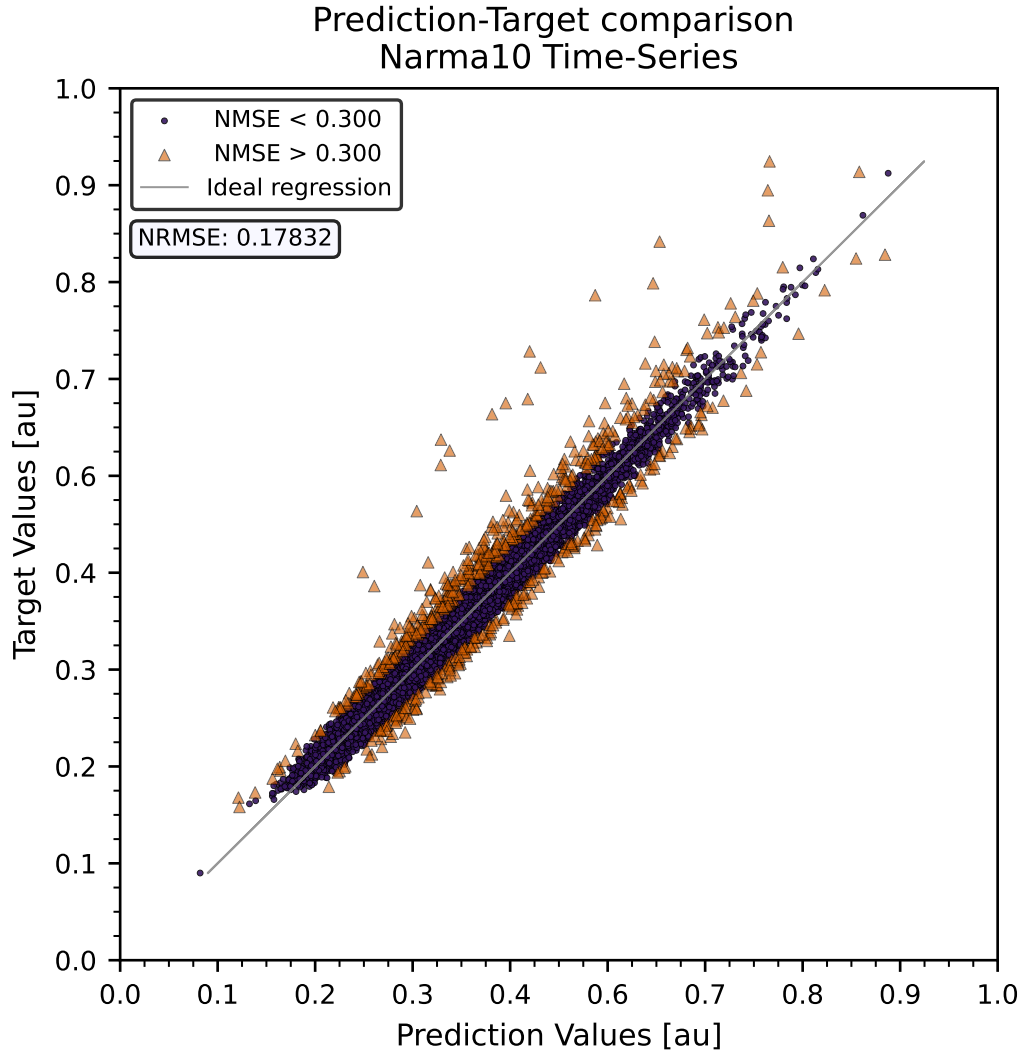


Figure C.23: Regression Plot. Comparison between the Narma10 and the mimicked systems obtained from the DLRC. Results from using FP numbers in a delay line with 800 VNs.

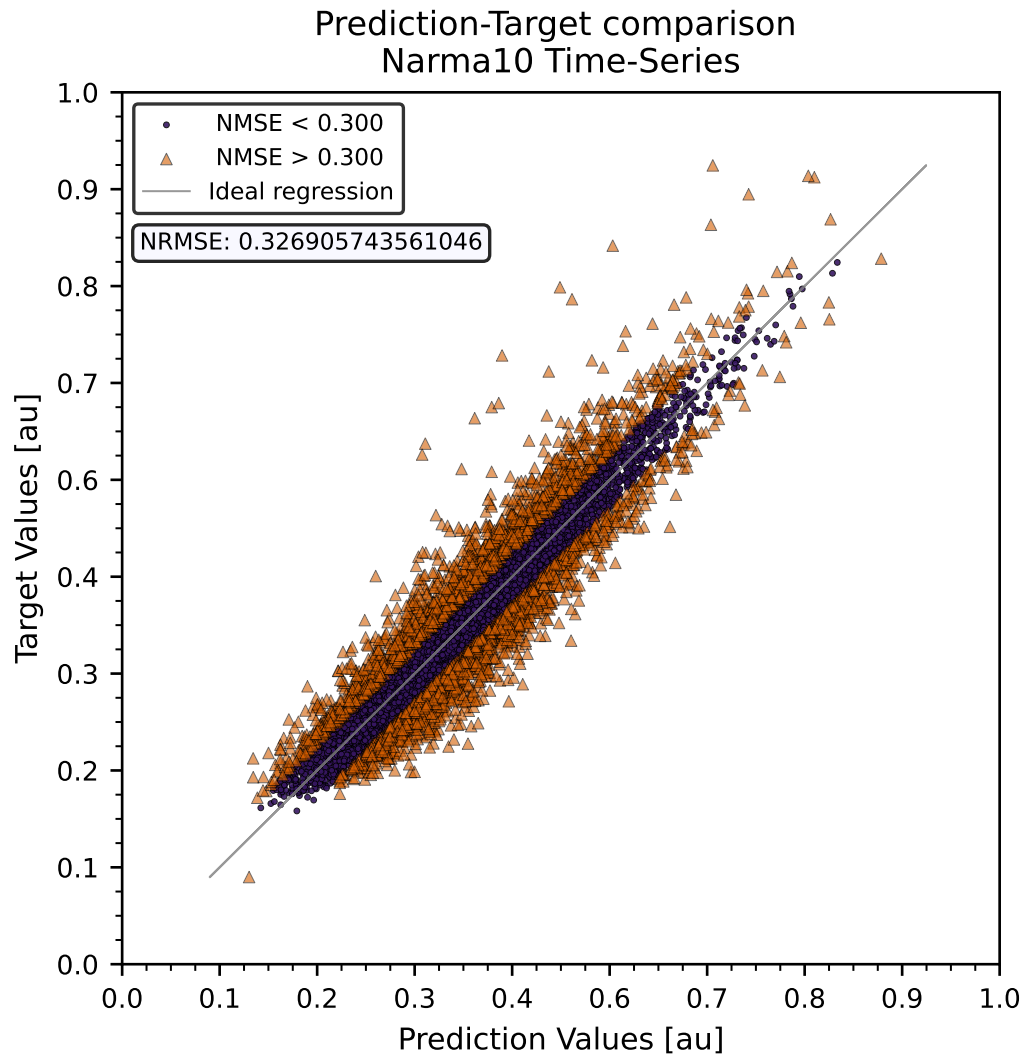


Figure C.24: Regression Plot. Comparison between the Narma10 and the mimicked systems obtained from the DLRC. Results from using FXP-Int16 numbers in a delay line with 800 VNs.

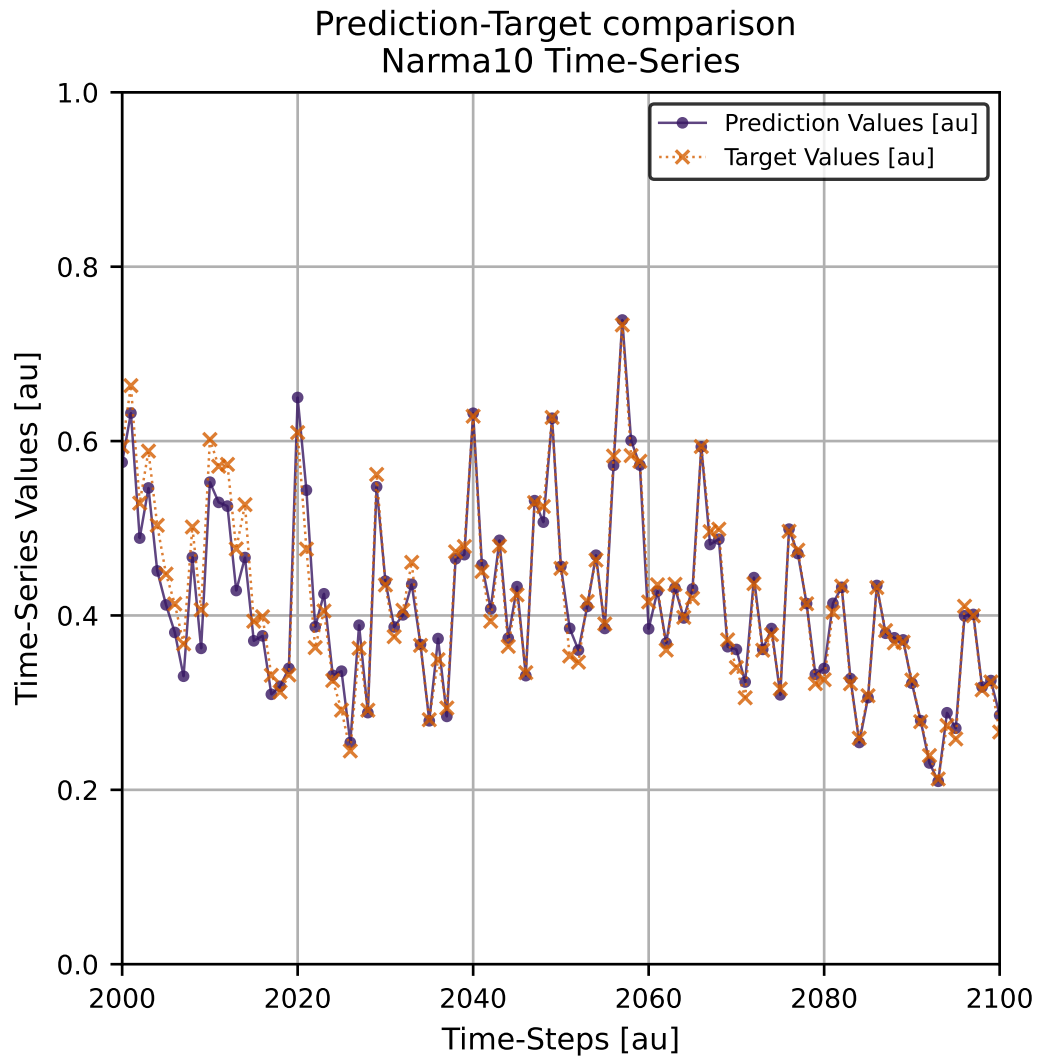


Figure C.25: Time Series plot. Comparison between the Narma10 and the mimicked systems obtained from the DLRC. Results from using FP numbers in a delay line with 800 VNs.

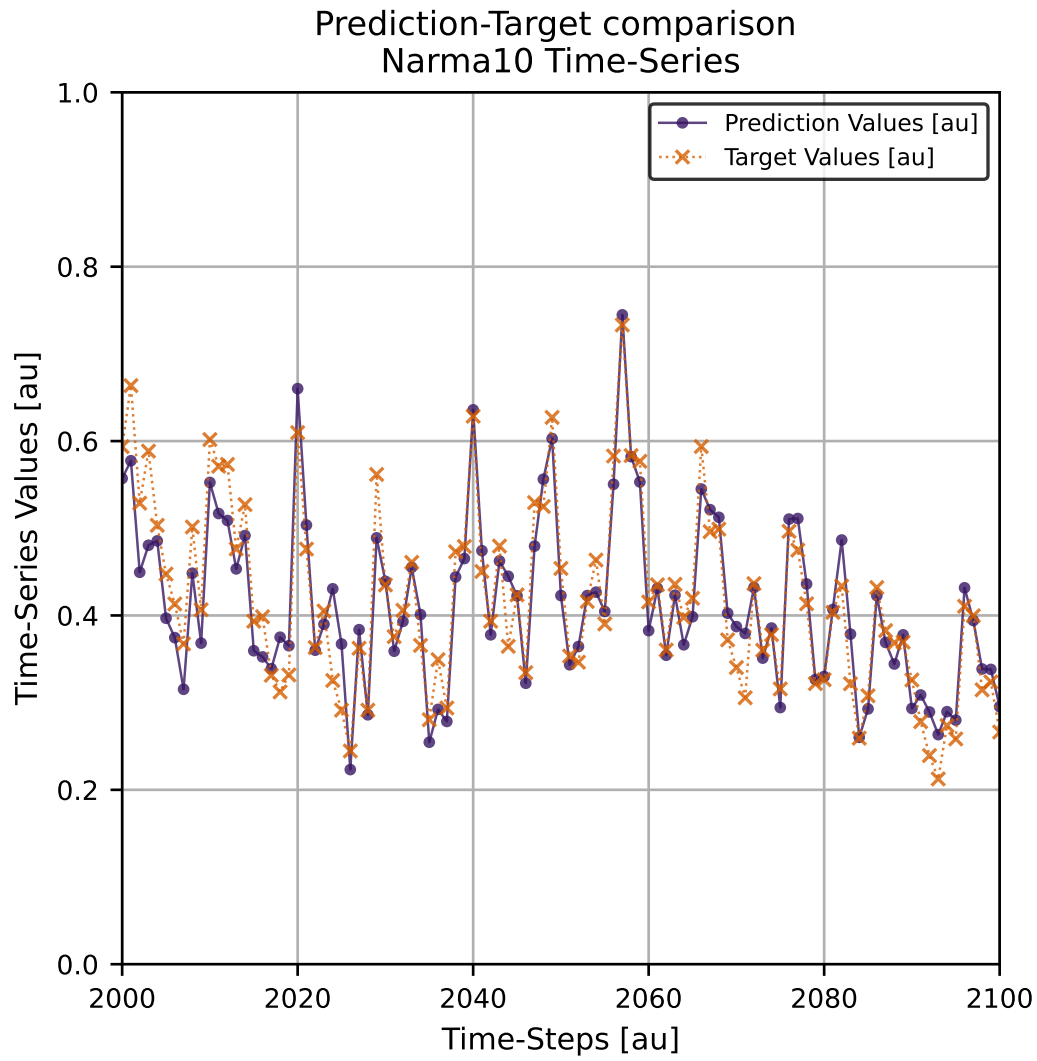


Figure C.26: Time Series plot. Comparison between the Narma10 system and the mimicked system obtained from the DLRC. Results from using FXP-Int16 numbers in a delay line with 800 VNs. .

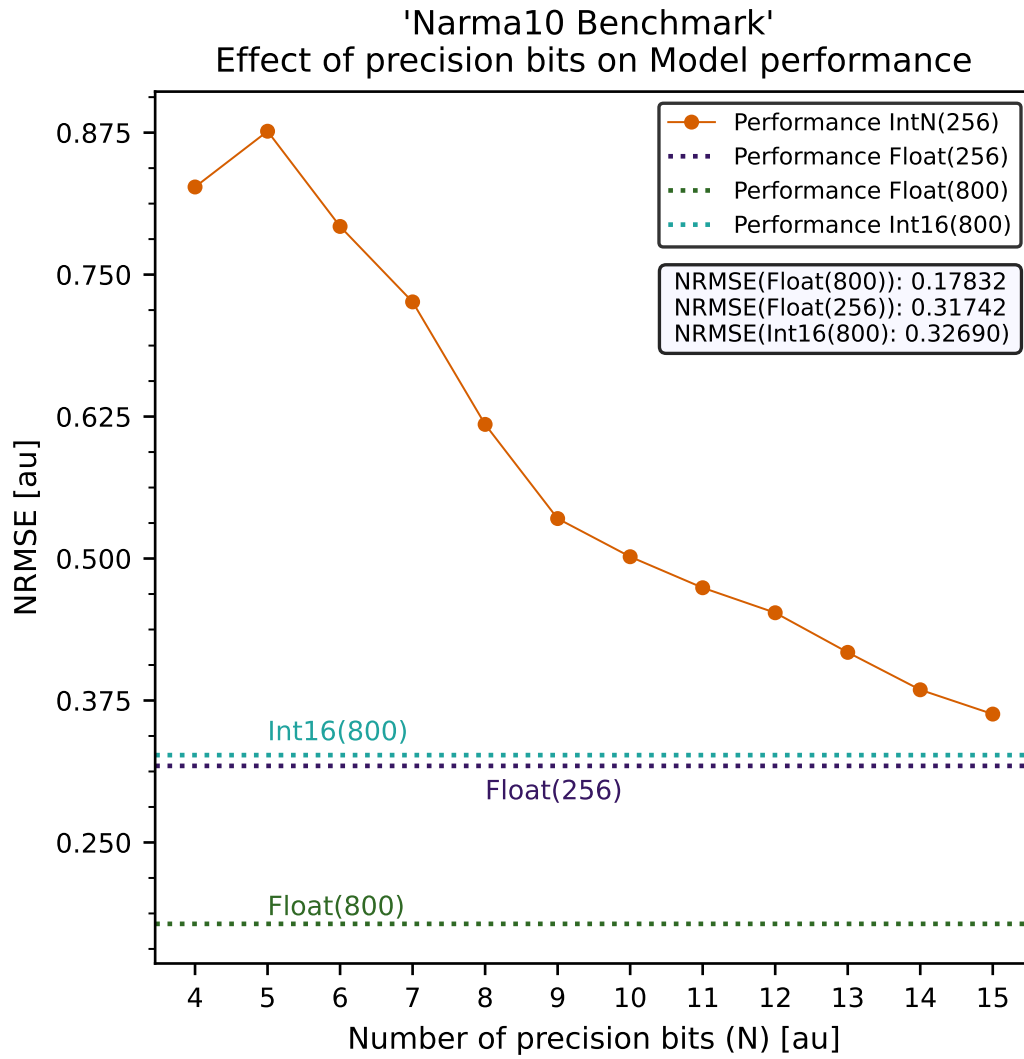


Figure C.27: Effects of number precision in the performance of the DLRC. Horizontal lines correspond to the best results from other experimental setups.

APPENDIX 4 - HARDWARE RESULTS

D.1 Mask Generator

D.2 Final

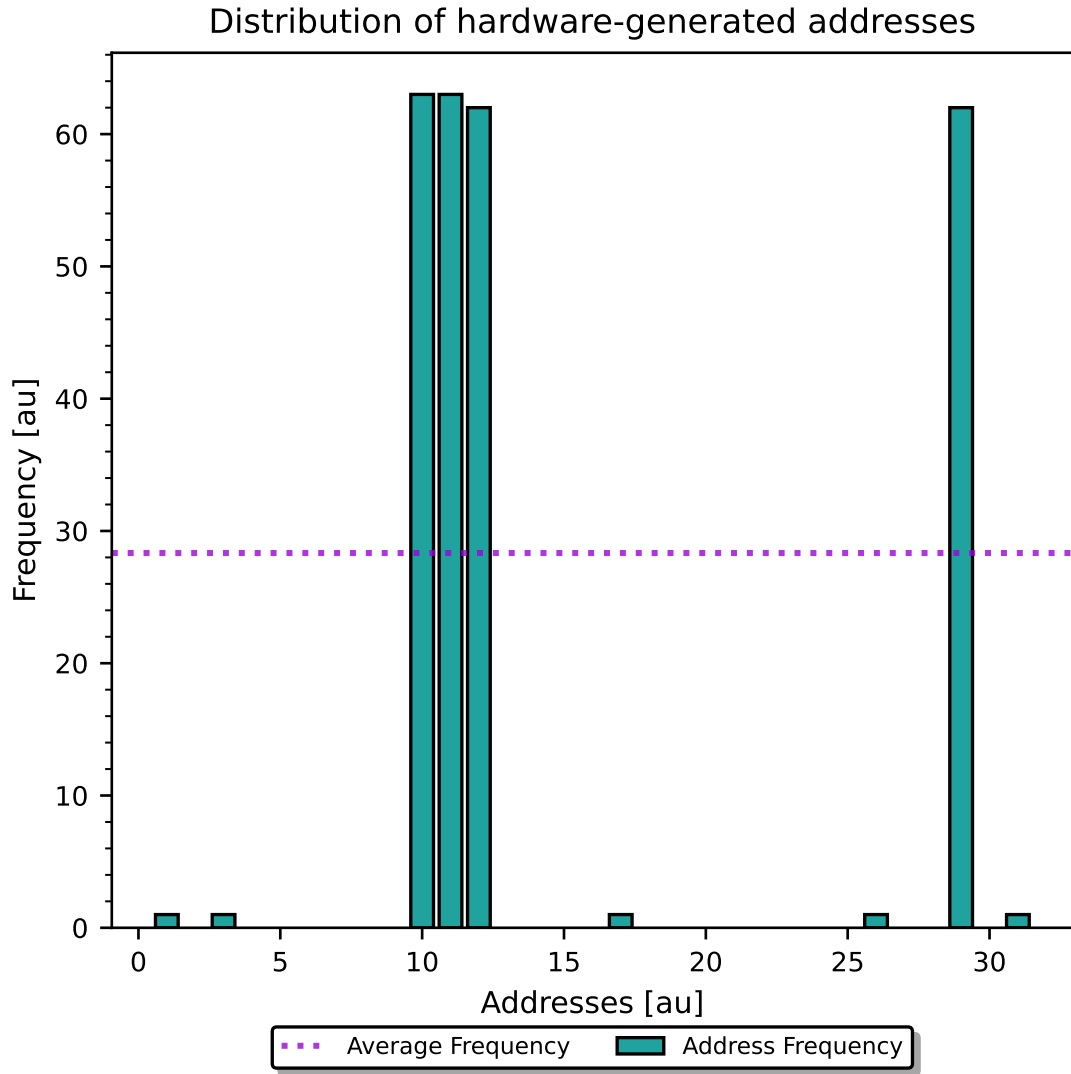


Figure D.1: Discrete distribution of generated mask addresses using a 5-LSB extraction. (From 8-bit FXP numbers) with the $x_0 = 0.3$

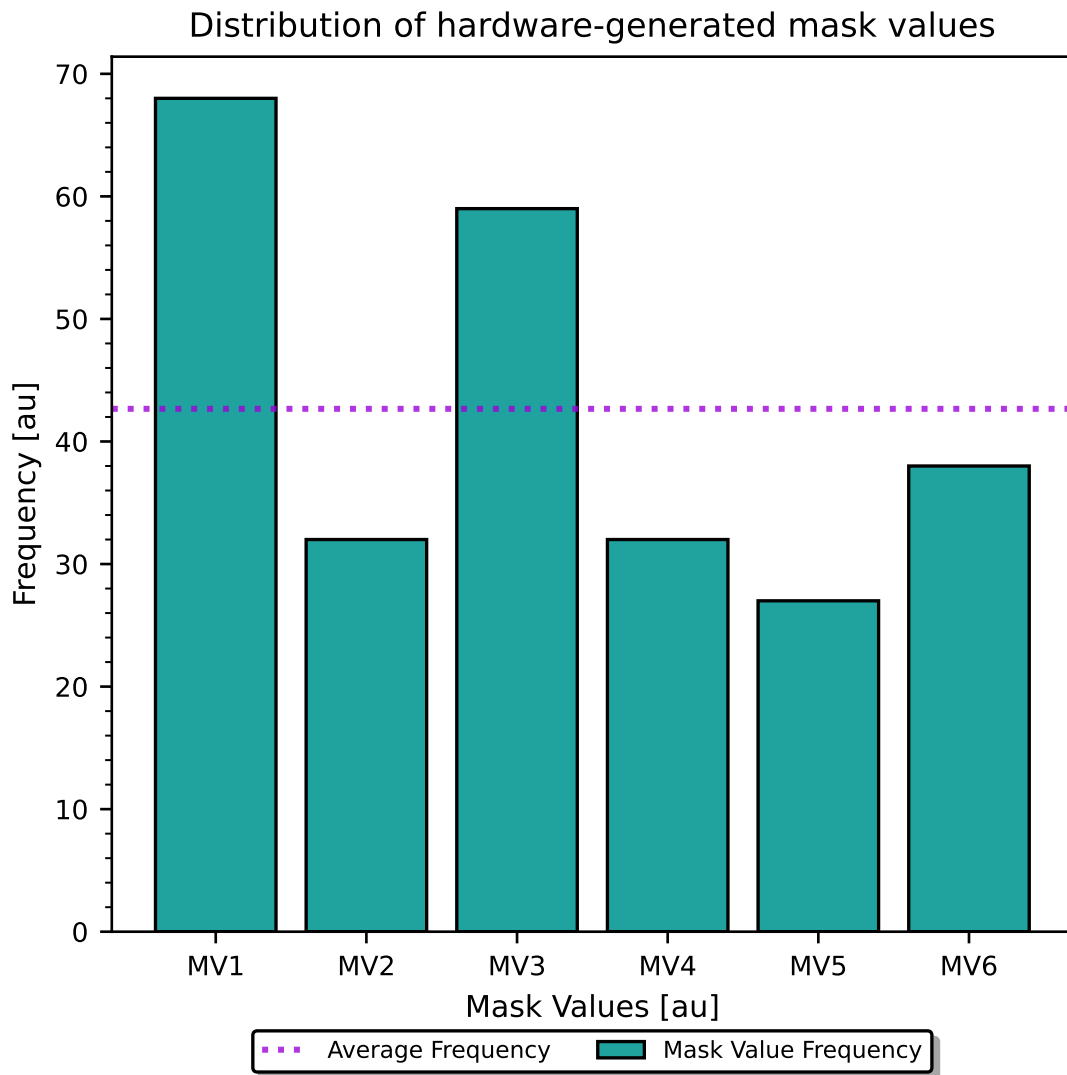


Figure D.2: The “Offset Bias” effect. Discrete distribution of mask addresses obtained using 3-LSB extraction over 255 iterations of the logistic map. Three extracted bits represent up to 8 registers. If we consider a six-valued mask, then two of its values — say MV1 and MV3 — have to fill the two remaining registers.

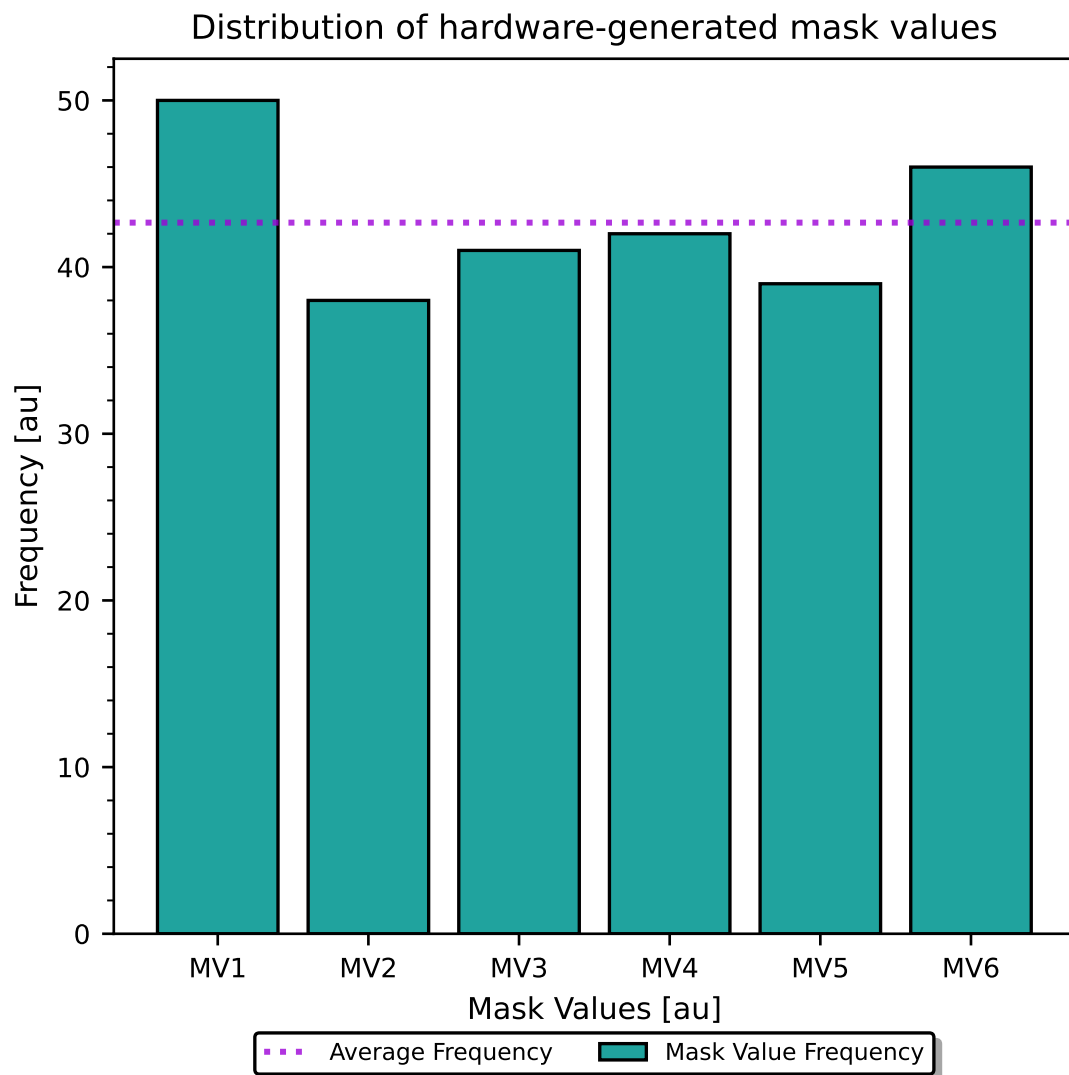


Figure D.3: Distribution of generated mask-values using a 5-LSB extraction. Compared to D.1 we are closer to a uniform distribution as the discrepancy between MV1, MV3 and the remaining mask values is less severe.

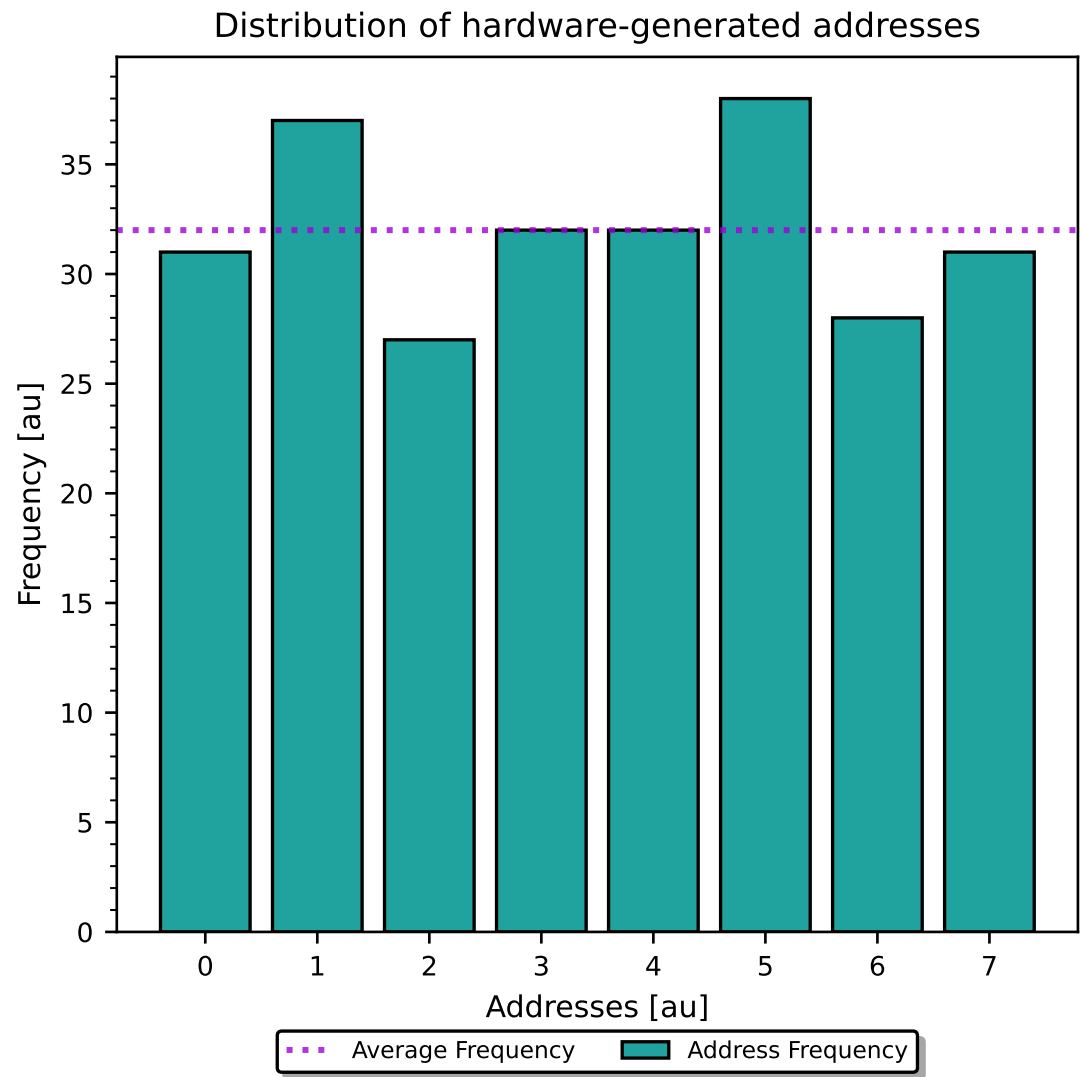


Figure D.4: Distribution of generated mask- addresses using a 3-LSB extraction.

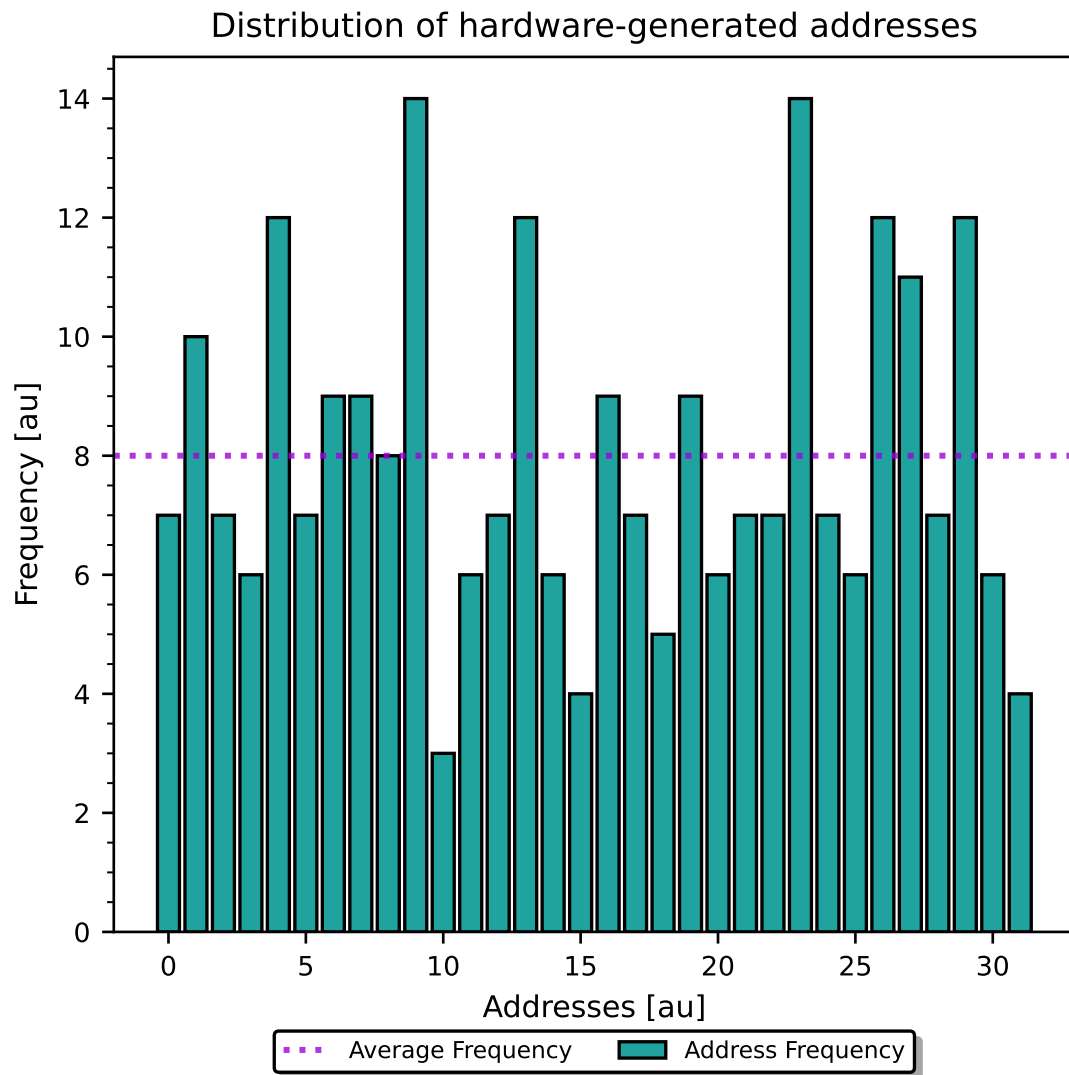


Figure D.5: Distribution of generated mask-addresses using a 5-LSB extraction.

Comparison between Twin Machine's Outputs (HW vs SW)

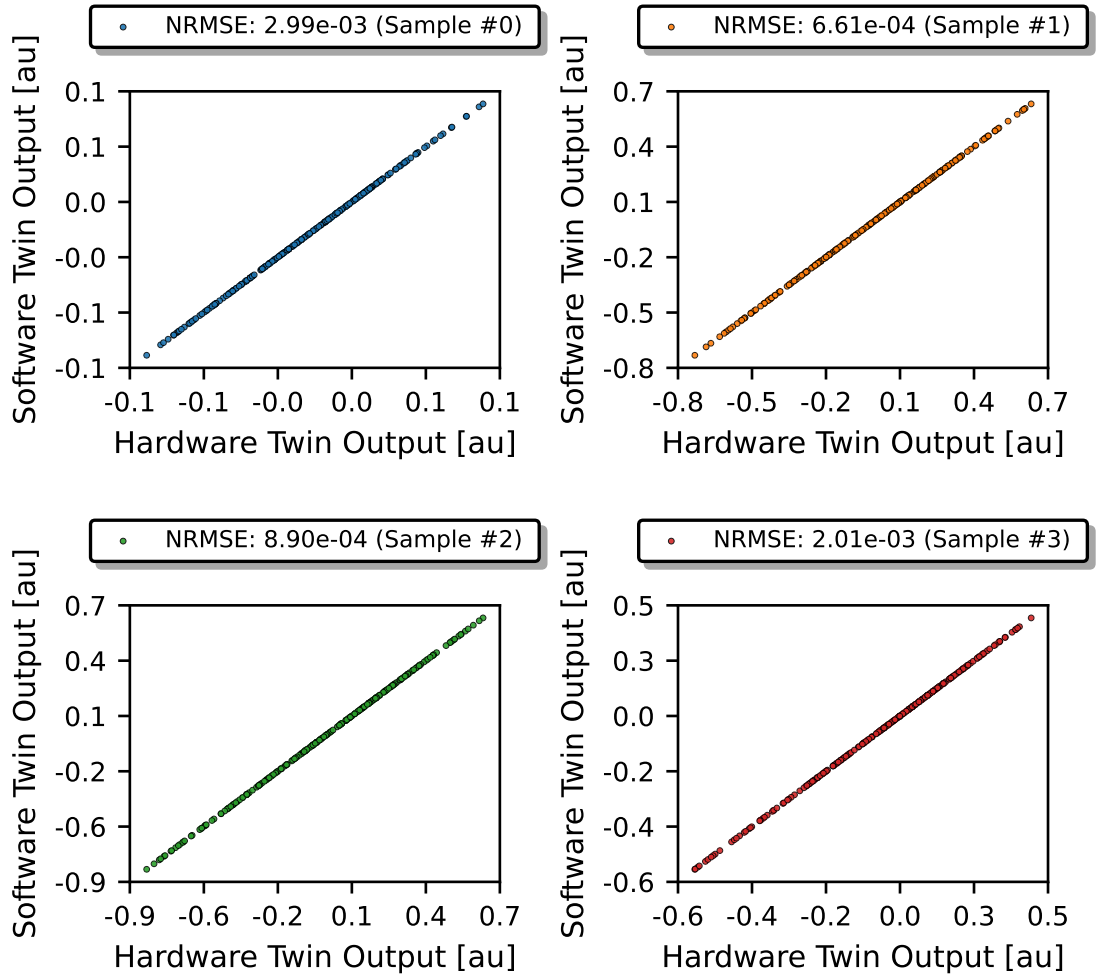


Figure D.6: Linear Regression comparing the output of the designed hardware with the target, the python model. The hardware retained a close match to the software, and for a multitude of samples.

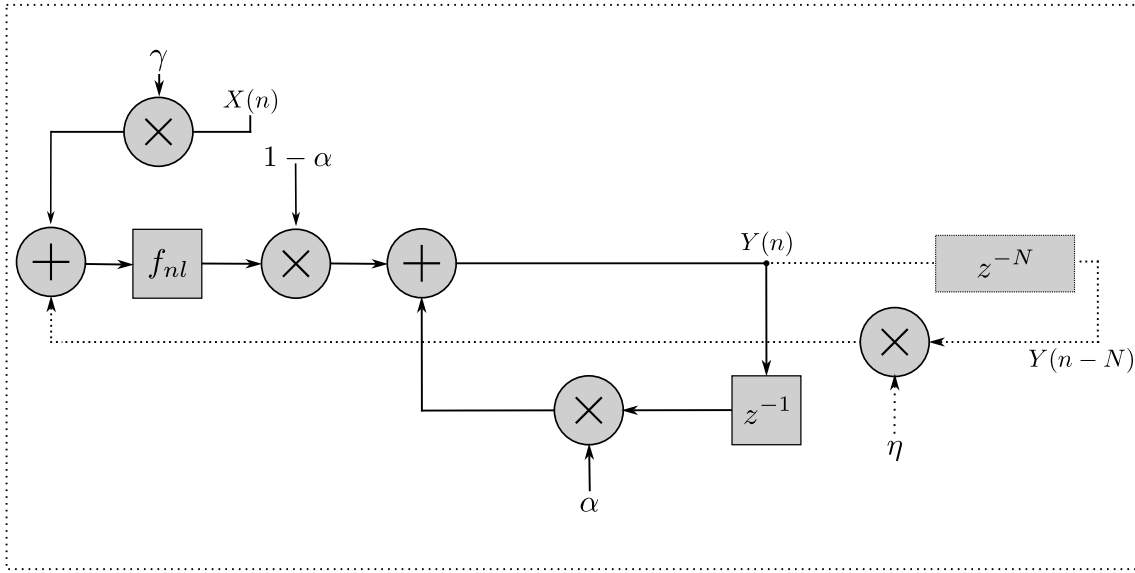


Figure I.1: Schematic of the Delay-Line Reservoir Computer. Solid lines highlight the discrete system's first-order impulse response filter (IIR) Adapted from [11].

$$Y[n] = a \cdot Y[n-1] + (1-a) \cdot f_{nl}(\gamma \cdot X[n] + \eta \cdot Y[n-N])$$

If we focus on the undotted part of the schematic of the delay line reservoir computer, the expression simplifies to:

$$Y[n] = a \cdot Y[n-1] + (1-a) \cdot f_{nl}(\gamma \cdot X[n])$$

This resembles a first order infinite impulse response (IIR) filter, as mentioned in [11]
Applying the Z-Transform

$$Y[z] - a \cdot Y[n-1] = (1-a) \cdot \gamma \cdot X[n]$$

$$Y[z] - a \cdot z^{-1} \cdot Y[z] = (1-a) \cdot \gamma \cdot X[z]$$

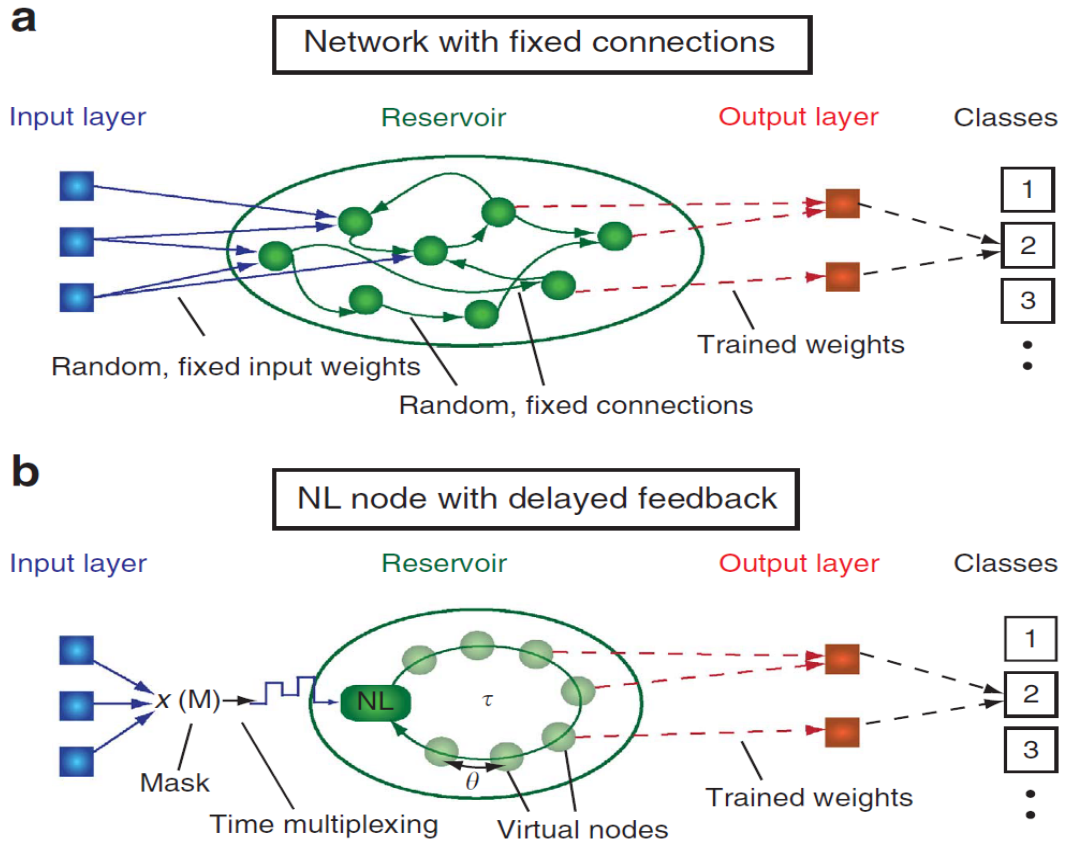


Figure I.2: Comparison between RC architectures. Classical (top), Delay-Line (bottom).

Imported from [2].

$$Y[z](1 - a \cdot z^{-1}) = (1 - a) \cdot \gamma \cdot X[z]$$

$$H[z] = \frac{Y[z]}{X[z]} = \frac{(1 - a) \cdot \gamma}{1 - a \cdot z^{-1}}$$

$$Y[z] - a \cdot z^{-1} \cdot Y[z] = (1 - a) \cdot \gamma \cdot X[z]$$

$$H[z] = \frac{Y[z]}{X[z]} = \frac{(1 - a) \cdot \gamma}{1 - a \cdot z^{-1}}$$

To find the pole(s) of this discrete filter:

$$X[z] = 0 \Leftrightarrow 1 - a \cdot z^{-1} = 0 \Leftrightarrow z = a$$





Day-Lin Research-Computer-System-Online-Research-Marketplace