FPGA-based Advanced Motion Controller Development and Design Automation

Gao, Yidan

Award date:
2022

EINDHOVEN UNIVERSITY OF TECHNOLOGY

ASM Pacific Technology

Department of Mathematics and Computer Science

FPGA-based Advanced Motion Controller Development and Design Automation

Master Thesis

Gao Yidan (1352806)

TU/e Supervisor:
Dip Goswami
ASMPT Supervisors:
Kai Wa Yan
Dragan Kostić
Mark Wijtvliet

version 11

Eindhoven, May 2022

Abstract

There is a popular trend to implement control systems on field-programmable gate array platforms for optimal hardware resources and fast computing speeds. Norm optimal iterative control is a high-complexity control algorithm which is hard to implement on a resource-constrained FPGA because it involves matrix operations. This project focuses on implementing a norm optimal iterative learning control algorithm on FPGA through programming with MATLAB and aims on providing an improved model with lower cost of lookup tables, flip-flops and digital signal processors.

Preface

I would like to thank Dip Goswami, my university supervisor, and Kai Wa Yan, my company supervisor, for giving me the opportunity to conduct research in collaboration with Eindhoven University of Technology (TU/e) and ASM Center of Competence. I'd like to thank Kai Wa Yan for his advice and assistance; whenever I asked for it, he always responded quickly, providing me with valuable advice and positive feedback, encouraging me to solve one problem after another throughout this project. I'd like to thank Dip Goswami for providing me with this opportunity to complete my graduation project after I contacted him and for his prompt assistance. I'd like to thank Dragan Kosti for his encouragement and for discussing issues with me, as well as for providing constructive feedback on control systems. I'd like to thank Mark Wijtvliet for his advice on hardware programming as well as his prompt feedback on my work. Finally, I'd like to thank my mother and father for their unwavering support of my study abroad plans, always supporting all my decisions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The complexity and performance of field-programmable gate arrays (FPGAs) have been growing exponentially. Due to its high loop rates (tens to hundreds of MHz) and its parallel execution model, the FPGAs are also increasingly being employed as a control system deployment platform [52]. In 2022, the FPGA chips in Kintex UltraScale+ series are equipped with up to 1.2M configurable logic blocks [17], far exceeding the XC2064, the first FPGA product [20]. The FPGAs consist of a series of grid blocks, which can be programmed. The designed circuits on the chips can be modified by designers to programme specific arithmetic operations. Even though CPUs have faster clock rates, FPGAs have higher throughput because FPGAs are designed for single functions that perform parallel calculations, whereas CPUs lack sufficient parallel processing support. Graphics processing units (GPUs) excel at parallel execution, but they are both expensive and power-hungry. Because of the additional delay on the logical block and inner routing, FPGAs, in general, are slower than GPUs. However, Shuichi Asano has shown that when all the processing units are busy with operations, FPGAs are able to perform comparable number of operations per time unit as GPUs [23]. In addition, the FPGAs consume much less power than GPUs when they perform the same function [34]. Application-specific integrated circuits (ASIC) are another option because they outperform GPUs and FPGAs in arithmetic operations. However, they are pricey and cannot be reconfigured like FPGAs. As a result, rather than being used during the design process, ASICs are typically used after the designed processes have been completed. Furthermore, FPGAs typically allow connection to almost any external device, whereas GPUs do not.

The FPGAs are programmed using hardware programming languages, eg, VHDL or Verilog, which require designers to be experienced in hardware programming and knowledgeable in the architecture of logical blocks and memories. Hardware programming languages require not only logical thinking but also basic knowledge of hardware circuits. Furthermore, these languages can handle not only sequential instructions but also concurrent executions. These characteristics make hardware languages more complicated than higher-level programming languages such as C. MATLAB allows designers to generate the hardware configurations without deep knowledge of VHDL or Verilog. MATLAB generates the hardware projects and interfaces based on Simulink models. When writing code, designers usually take a long time to debug while automatic code generation would have almost no syntax error but achieve the same function. Furthermore, Simulink provides many models on different levels, from lookup tables (LUTs), the basic logical units in the FPGAs, to User-Defined Functions which could be customized by designers. As a popular and common tool, MATLAB and Simulink are useful to designers, e.g. control engineers, who are often not skilful in hardware programming.

Control systems are widely applied in modern society from electrical devices in daily life to high-speed, high-precision machines in industries. Actuators in control systems are typically expected to maintain a stable state or to accurately meet requirements. A control system is designed to control machines or systems to fulfil requirements [43]. The most popular controller is propor-

tional–integral–derivative (PID) controller, which was first mathematically analysed by Nicolas Minorsky (1922) based on his observation of how helmsmen steer, which is based on the current direction error, the error in the past, and the rate of change of error [58]. Nowadays, actuators which require high precision are mostly controlled by feedback systems, especially in high precision industries. A feedforward signal can be added to reduce error. Feedforward control systems can be designed based on prior knowledge of the control target [59].

Iterative learning control (ILC) is first proposed by Murray Garden in 1967 [42] and it became widely known since 1984. The ILC algorithm generates a feedforward signal to improve control performance [61]. When a system performs the same task repeatedly, it is assumed that the error is also repetitive. ILC is able to trace this error and reduce it to smaller errors after some iterations. Thus, for a repetitive input, ILC helps systems to achieve higher accuracy and reduce settling time. Gunnarsson and Norrlöf minimizes the ILC algorithm in 2001, where the input does not have to be repetitive, allowing a small variation for motion tasks [45], Jin (2018) provided a novel ILC algorithm permitting iteratively varied input and random initial position [50]. Norm-optimal iterative learning control (NOILC) is one extension of ILC algorithms which optimizes the 2-norm of error [63]. It consists of two filters represented as two-dimensional matrices whose sizes are dependent on the number of data samples in each iteration [75]. These two filters are also designed by adjusting three weighting matirces to achieve different targets [27][72]. The output feedforward signal calculation for the next iteration is based on filtering the error signal and the previous feedforward signal.

Including ILC, there exist many control algorithms that use matrix operations in their calculation such as quadratic dynamic matrix control [65], dynamic matrix control [35], linear quadratic regulator control [41], and multiple-input-multiple-out PID control algorithms. Because matrices operations are frequently of high complexity, these control algorithms with matrix operations frequently encounter resource issues while implemented on the FPGA platforms. As a result, they are usually implemented on computers with a lot of memory, however, the real-time ability of a control system may lose by CPU implementation. Due to the time limitation, even though there exist other similar algorithms, in this project only NOILC is implemented.

The main purpose of this project is to implement the ILC algorithm on an FPGA. Besides the implementation, the resource utilization for the ILC model is reduced, allowing a system with a large number of samples to be implemented on the chosen FPGA and the resource utilization is predicted before the final implementation by a proposed model.

Chapter 2

Problem Statement

Because of the high computational complexity of the matrix multiplication in NOILC algorithm, when the scalar of an NOILC is larger, like sampled for more than hundreds of or even thousands of times for one tried signal, the implementation of NOILC requires huge resources including memory and logic units. However, FPGAs are usually not designed with huge resources. There do exist FPGAs with a large number of resources but they are typically more expensive. Thus, implementing NOILC on a relatively cheap FPGA is a challenge. In addition, the actual resource utilization is not provided until the final implementation when the HDL code is generated by MATLAB. As a result, generating a complete project every time the designer wanted to know the amount of resources used would be extremely time consuming. To solve these problems, there are two main contributions in this thesis:

1. The improved NOILC algorithm which utilizes fewer resources on the board but achieves the same functions.
2. The resource model for the improved and classical NOILC algorithms to estimate the resource utilization in the design process in MATLAB.

Chapter 3

Related work

3.1 Computation load reduction of NOILC

The ILC algorithm cannot avoid utilizing two square matrices as filters for computation. Because of its high complexity $\mathcal{O}(n^2)$ [30], the computation load would increase sharply with the rising in the number of samples. With assuming the number of samples to be N, $2 \times (N^2 + N)$ elements joint the computation and $2 \times N^2$ multiplications are required to complete the NOILC algorithms [30]. Thus, for a large project, the required resource would increase to an affordable number.

A recent study on decreasing the computation load for norm-optimal ILC algorithm for both linear time-invariant (LTI) and linear time-varying (LTV) systems is published in 2016 [75] by Zunder, Bolder, Koekebakker and Oomen. They propose a resource-efficient ILC reducing the computational load from $\mathcal{O}(N^3)$ to $\mathcal{O}(N)$ by exploiting state-space descriptions and inversion techniques. They have shown that the new approach is able to generate the same control signal with a much lower computational cost. Moreover, this approach has been successfully executed on a host computer with a large task where the number of samples is 100000 by the same group [74]. Unfortunately, in this approach, its state-space update in real-time may cost much more power than the traditional norm-optimal ILC algorithm since not only the input signal, position error, is transmitted from ARM to FPGA but also the updated state-space. For every sample, the space-state has to be recomputed. In addition, a part of the calculation is inverse, from time N to time 1, providing difficulty in the calculation. When compared to the method proposed in this report, much more power is consumed because more data is communicated between the ARM and FPGA, and it is more complex, making implementation more difficult.

3.2 ILC Implementation on FPGA

Fei et al. have implemented a combination of model predictive control (MPC) and ILC in a digital signal processing/field programmable gate array (DSP-FPGA)-based azimuth axis platform of a telescope to control a permanent magnet synchronous motor [39]. The propose of this paper is to minimize the speed ripple. The reason why they choose ILC algorithms is to suppress periodic torque pulsations because the onset of torque pulsation in permanent magnet synchronous motor control systems is primarily related to rotor position, which results in periodic oscillations in motor torque and speed. In this paper, they do not improve the ILC algorithm, and the resource used is not mentioned. Qiu et al. have implemented a combined disturbance-observer-based control and iterative learning control design on an FPGA platform for pulsed superconducting radio frequency cavities [64]. They demonstrated that combining the ILC algorithm speeds up the system's convergence. Similarly, they did not mention of the ILC's resource consumption or the algorithm's improvement. They emphasized on the system's convergence speed.

Awan in 2012 implemented the NOILC algorithm on an FPGA applying it to a Gantry robot proving that the calculation time is reduced from 830ms (implemented on the FPGA) to 1.47ms (implemented on software) [24]. The FPGA-based implementation is several hundred times faster than the software. However, the controller requests a large storage space to store the trial data, the input signal for each iteration, which is restricted by the design and cost of chips. The authors have not improved the storage or resource utilization of NOILC during the implementation. They mentioned that the limitation of memory and power should be solved in the future while the aim of this thesis is to solve the restriction of resources.

3.3 Matrix Multiplication

3.3.1 Timing and Power Improvement

Considering that in most situations, ILC cannot avoid matrix operation. Matrix multiplication is the most expensive computation in ILC. Thus, decreasing the resource utilization would be helpful for ILC implementation. Many previous works on implementing matrix multiplication on an FPGA have focused on reducing latency and power consumption [48][22][56]. To achieve shorter latency or less power consumption it always sacrifices some area which means higher resource utilization [48]. There are exceptions, however, and the methodology proposed by Campbell and Khatri in 2006 can reduce not only the computation time but also the resources utilization [31].

Parallel computing is that multiple computations or processes are executed at the same time by employing more than one processing units. It is effective in reducing computational latency [47]. The basic idea is to use multiple processors to collaboratively solve the same problem, i.e. the problem to be solved is broken down into several parts, each of which is computed in parallel by separate processors.

Pipelining is parallel execution at the instruction level which is a significant method in performance optimization especially in decreasing the execution time for a loop which repeats one instruction many times [21] which is taken off in 1969 [44]. Pipelining results in quasi-parallel implementation of a program in which multiple instructions are simultaneous executed. Because pipelining increases the throughput by executing multiple data-independent instructions at the same time, it is not efficient if only one instruction is executed. But for multiple instructions who do not read the output data of other instructions when they are simultaneous executed, using the parallelism principle of pipelining can actually increase the throughput by several times. Each pipeline stage has its own combined logical datapath within it, with no multiplexing of resources between them, so the area overhead is relatively high. But the effect of pipelining is achieved by allowing different pipeline stages to do different things at the same time, improving performance, optimising timing and increasing throughput rates.

Pipelining is helpful in increasing the throughput but the crucial restriction in ILC implementation is that the resource on a FPGA may not enough for ILC implementation.

3.3.2 Resource Utilization Optimization

Fox, Otto and Hey 1987 proved that decomposing matrices into smaller square sub-matrices is efficiently matrix multiplication [40]. The PUMMA algorithm proposed by Choi et al. in 1994 extended the Fox algorithm to a two-dimensional block curtain data distribution. It solves the problem of GPU based matrix multiplications where the sizes of matrices are usually larger than the shared memory of GPUs [33]. In 1995 van de Geijn and Watts proposed a simpler and more efficient matrix product algorithm called SUMMA [70] which achieves overlapping between com-

puting and communications. Different from CANNON [32] which restricts that the matrices have to be a square matrix with the same size, SUMMA allows arbitrary sizes of matrices to do multiplication operations. This algorithm has been enhanced to support 64-bit floating-point FPGA matrix multiplication (Dou, Vassiliadis, Kuzmanov & Gaydadjiev, 2005) [37] which is acceptable for arbitrary matrix sizes [70]. During the calculation, matrices are split into sub-matrices and parallelly process these sub-matrices. By doing so, matrices with large sizes are able to be implemented on a resource-constrained platform and shorten the computation time from sequential block matrix multiplication. The floating-point operations per second in this design [37] are at least 1.7 times faster than the related design in [69] and up to 18 times faster than the design in [66], assuming they implemented on the same chip.

Lin, So and Leong (2011) have shown that dense and sparse matrix multiplications based on an FPGA are restricted by different factors. The sparse matrix multiplication is constrained by the input/output memory while the computation of dense matrices is limited by the computation limitation [57]. Kestur, Davis and Chung provided a new coding encoding method in 2012 which decreases the memory accesses by 25% on average compared with compressed sparse row format for the sparse matrix examples [53]. The storage requirement is decreased significantly due to the new encoding method, however, the problem of limited computational units cannot be solved by it.

Decomposing matrices into smaller sub-matrices is helpful in implementing ILC algorithms because it decrease the memory accesses and restricted the usages of multipliers which are the goal of this project. The method proposed in this report also decomposes the matrices into columns for calculations based on the particularity of matrix-vector multiplication.

3.4 High-Level Synthesis

High-Level Synthesis (HLS) is an automated design process that takes an algorithm description as input to create the digital hardware that implements the required functionality [62]. Control algorithms are typically written in higher-level programming languages such as C, but the platforms that run these control algorithms frequently require HDLs. High-level programming languages are frequently more user-friendly than hardware languages. As a result, when using FPGAs or ASICs for data path implementation, HLS has emerged as an alternative to HDL[28]. Although HLS allows for rapid prototyping of any stochastic algorithm, there are limitations in terms of performance, memory bandwidth, and the number of logics when compared to manual design by domain experts [54]. Considering that HLS is able the partly restrict the performance of HDL codes by instructions in the high-level languages which MATLAB cannot achieves. The codes generated by HLS may has higher performance on latency, area, power, etc. However, MATLAB has a distinct advantage in that the designer can view the entire system model more intuitively.

3.5 FPGA programming by MATLAB Simulink

Haldar et al. (2001) used MATLAB to assist in the generation of HDL code, reducing design time from days to minutes [46]. Siwakoti and Town have designed FPGA-based digital controllers in MATLAB Simulink [67]. Different from the traditional manual HDL coding method, they generate the code by using the HDL generator provided by MATLAB. Jiang and Mangharam (2013) have utilized MATLAB and Simulink to create the first-of-its-kind electrophysiological model of the hear. They generated the VHDL code of this model directly through MATLAB and Simulink and implemented it on the real-time hardware for testing. The generated VHDL code is efficient and helps them to achieve the implementation of multiple versions of the model on the FPGA [49]. Weinmann created embedded MEDUMAT Transport software for healthcare workers in 2014 by creating the model in MATLAB and generating the HDL code through MATLAB after verifying

the simulated result was correct. He found that with the help of MATLAB, the code development and review were accelerated by 50%, meanwhile more versions of the model are developed [38]. When designers are unfamiliar with hardware languages, FPGA programming through MATLAB has been shown to be easier for designers to implement models and may be more efficient than traditional programming methods in the design process. In terms of resource utilization, latency, energy consumption and etc., no evidence has been found that MATLAB-generated HDL code outperforms hand-written code by professional engineers. FPGA programming by MATLAB is more helpful for those who are not familiar with hardware languages and try to implement an operational hardware project.

Chapter 4

Proposed approach

The NOILC contain two matrix multiplications which are used to filter the input signal, position error, and the feedforward signal from the current iteration. Due to the high complexity of the matrix multiplication and other matrix operations in the NOILC, NOILC is hard to be implemented on the FPGA.This chapter aims to introduce two methods to implement the NOILC algorithm on the FPGA and predict the resource utilization of these two methods.

4.1 Preliminary

An arbitrary matrix named A with m rows and n columns is represented in bold, $\mathbf{A} \in \mathbb{R}^{m \times n}$. One element in the k^{th} row ($1 \leq k \leq m$) and l^{th} column ($1 \leq l \leq n$) of matrix $\mathbf{A}$ is indicated as $A_{k,l}$. The k^{th} row of matrix $\mathbf{A}$ is 1 by N vector $A_{r,k}$ and the l^{th} column of matrix $\mathbf{A}$ is N by 1 vector $A_{c,l}$.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A_{c,1}} & \mathbf{A_{c,2}} & \dots & \mathbf{A_{c,l}} & \dots & \mathbf{A_{c,n}} \end{bmatrix} \tag{4.1}$$

$$= \begin{bmatrix} \mathbf{A_{r,1}} \\ \mathbf{A_{r,2}} \\ \vdots \\ \mathbf{A_{r,k}} \\ \vdots \\ \mathbf{A_{r,m}} \end{bmatrix} \tag{4.2}$$

$$= \begin{bmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,l} & \dots & A_{1,n} \\ A_{2,1} & A_{2,2} & \dots & A_{2,l} & \dots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ A_{k,1} & A_{k,2} & \dots & A_{k,l} & \dots & A_{k,n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \dots & A_{m,l} & \dots & A_{m,n} \end{bmatrix} \tag{4.3}$$

The NOILC system implemented in this project is a single-in, single-out (SISO) linear control system. The design is in a discrete-time domain where the sample period for the input and output are both Ts and the sampling frequency is $f_s = 1/Ts$. For the NOILC system in this project, the input of the system is a repetitive signal. In this project, the repetitive signal is the reference position signal. This signal is repeatedly entered until the system stops. Each repetition is called as one iteration. The length of the input signal in one iteration is assumed as N, where $N \in \mathbb{Z}^+$. The vector constructed by all samples of an arbitrary signal s in iteration j is $\mathbf{s_j} \in \mathbb{R}^N$ where the i^{th} sample of the signal is the i^{th} element of the vector. The i^{th} sample of the signal s with length

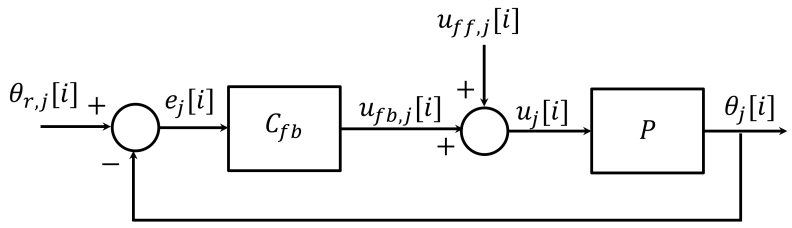

Figure 4.1: Block diagram of basic feedback control system with a feedforward signal

N in the j^{th} iteration is denoted as $s_j[i]$, $i = 1, 2, \ldots, N$. Iterations are indexed as a positive integer j without an upper bound considering that a system can run forever, $j \in \{1, 2, \ldots \infty\}$.

$$\mathbf{s_j} = \begin{bmatrix} s_j[1] \\ s_j[2] \\ \vdots \\ s_j[i] \\ \vdots \\ s_j[N] \end{bmatrix} \tag{4.4}$$

Figure 4.1 depicts a classic feedback control model in the discrete-time domain. The feedback controller is C_{fb} in the z-domain and the plant is P also in z-domain. The input signal of the feedback control is the reference position ($\theta_{r,j}[i]$) and the output is the encoder measurement ($\theta_j[i]$). The reference position is the expected trace of the actuator and the encoder measurement is the measured position of the actuator. The signal position error, $e_j[i]$, is the difference between $\theta_{r,j}[i]$ and $\theta_j[i]$, calculated by Equation 4.5. The output signal of the feedback controller is called $u_{fb,j}[i]$, the feedforward signal is $u_{ff,j}[i]$ and the sum of them is the control signal $u_j[i]$. The reference position has a finite length, and the length for each execution remains constant. The goal of this control system is to minimize the position error, $e_j[i]$. As the input signal of the controller, $e_j[i]$ is calculated after the signal $\theta_{r,j}[i]$ input to the system from the outside. The output signal of the feedback controller, $u_{fb,j}[i]$, can be added with a feedforward signal $u_{ff,j}[i]$ to help with speeding up the convergence. The sum of them is the control signal $u_j[i]$ which will be applied to the plant. The output of the plant is measured to compute the current position error forming a close loop.

$$e_j[i] = \theta_{r,j}[i] - \theta_j[i] \tag{4.5}$$

4.2 Norm-optimal Iterative Learning Control

The input signal of a single-in-single-out NOILC in this project is repetitive. The input signal in this thesis is the position error e_j from the j^{th} iteration, and the output signal is the feedforward signal $u_{ff,j+1}$ which will be added to the feedback control signal $u_{fb,j+1}$ in the $j+1^{th}$ iteration. In the discrete-time domain, both input and output are one-dimensional signals. The sample periods of these two signals are both T_s. In each iteration, the number of output samples should be the same as the input samples which is the scalar of the input, $N \in \mathbb{N}^+$. The aim of this NOILC system is to generate a feedforward signal helping the control system to decrease the error and speed up the convergence.

L is a learning filter and Q is a robustness filter [29]. For simplicity, two matrices are two N by N matrices which are calculated offline, $\mathbf{L}, \mathbf{Q} \in \mathbb{R}^{N \times N}$. The values of these two matrices are based on three weighting matrices, $\mathbf{W_e}, \mathbf{W_f}$ and $\mathbf{W_{\Delta f}}$ [29] [72] and the impulse response $\mathbf{PS}$ of the close-loop from $u_{ff,j}$ to θ_j in Figure 4.1. $\mathbf{W_e}$ is set to determine the weighting of error.

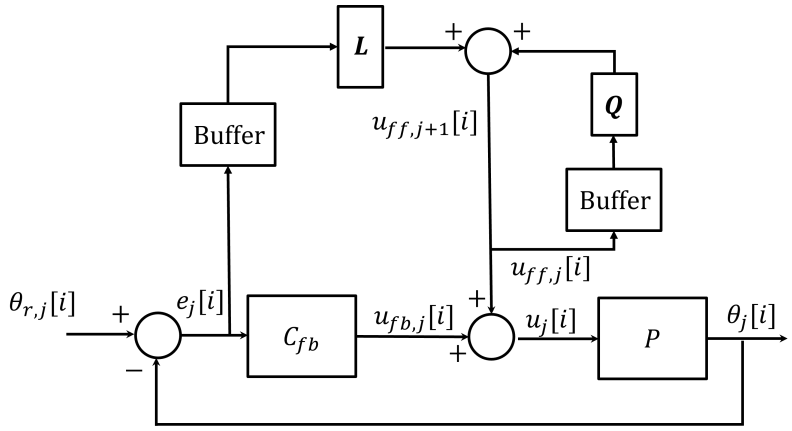

Figure 4.2: Block diagram of ILC control [73]

$\mathbf{W_{\Delta f}}$ controls the convergence speed and the sensitivity to iteration varying distributed and $\mathbf{W_f}$ influence the robustness with respect to model uncertainty [29]. In discrete-time domain, the process sensitivity function in Equation 4.6 can be represented by convolution matrix **PS** which is constructed by the impulse response data $PS(i)$ of the close-loop system from $u_{ff,j}$ to θ_j, where i is time, $i = 0, 1, \ldots N-1$ [26].

$$\frac{\theta_j(s)}{u_{ff,j}(s)} = PS(s) = \frac{P(s)}{1 + P(s)C_{fb}(s)} \tag{4.6}$$

$$\mathbf{PS} = \begin{bmatrix} PS(0) & & 0 \\ \vdots & \ddots & \\ PS(N-1) & \ldots & PS(0) \end{bmatrix} \tag{4.7}$$

With assuming the input and output as $\boldsymbol{\theta_j}$ and $\mathbf{u_{ff,j}}$, the response of the system is in Equation 4.8 and Equation 4.9, where $\boldsymbol{\theta_j}$ and $\mathbf{u_{ff,j}}$ are vectors composed by signals θ_j and $u_{ff,j}$ in the j^{th} iteration.

$$\boldsymbol{\theta_j} = \mathbf{PS} \cdot \mathbf{u_{ff,j}} \tag{4.8}$$

$$\begin{bmatrix} \theta_j(0) \\ \cdots \\ \theta_j(N-1) \end{bmatrix} = \begin{bmatrix} PS(0) & & 0 \\ \vdots & \ddots & \\ PS(N-1) & \ldots & PS(0) \end{bmatrix} \begin{bmatrix} u_{ff,j}(0) \\ \cdots \\ u_{ff,j}(N-1) \end{bmatrix} \tag{4.9}$$

The objective is to find the $\mathbf{u_{ff,j}}$ with the minimizes the cost function $\jmath$ in Equation 4.10 [73].

$$\jmath = \mathbf{e_{j+1}^T W_e e_{j+1}} + \mathbf{u_{ff,j+1}^T W_f u_{ff,j+1}} + (\mathbf{u_{ff,j+1}} - \mathbf{u_{ff,j}})^T \mathbf{W_{\Delta f}} (\mathbf{u_{ff,j+1}} - \mathbf{u_{ff,j}}) \tag{4.10}$$

With the knowledge $\mathbf{e_{j+1}} = \mathbf{e_j} - \mathbf{PS}(\mathbf{u_{ff,j+1}} - \mathbf{u_{ff,j}})$, the cost function can be rewrite as

$$\begin{aligned} \jmath =& [\mathbf{e_j} - \mathbf{PS}(\mathbf{u_{ff,j+1}} - \mathbf{u_{ff,j}})]^T \mathbf{W_e} [\mathbf{e_j} - \mathbf{PS}(\mathbf{u_{ff,j+1}} - \mathbf{u_{ff,j}})] \\ &+ \mathbf{u_{ff,j+1}^T W_f u_{ff,j+1}} + (\mathbf{u_{ff,j+1}} - \mathbf{u_{ff,j}})^T \mathbf{W_{\Delta f}} (\mathbf{u_{ff,j+1}} - \mathbf{u_{ff,j}}). \end{aligned} \tag{4.11}$$

When the cost function is minimized, $\frac{\partial \jmath}{\partial \mathbf{u_{ff,j+1}}} = 0$. The optimal feedforward signal in the $j+1$ iteration is represented as

$$\mathbf{u_{ff,j+1}} = [(\mathbf{PS})^T \cdot \mathbf{W_e} \cdot \mathbf{PS} + \mathbf{W_f} + \mathbf{W_{\Delta f}}]^{-1} \{(\mathbf{PS})^T \cdot \mathbf{W_e} \cdot \mathbf{e_j} + [(\mathbf{PS})^T \cdot \mathbf{W_e} \cdot \mathbf{PS} + \mathbf{W_{\Delta f}}] \mathbf{u_{ff,j}}\}. \tag{4.12}$$

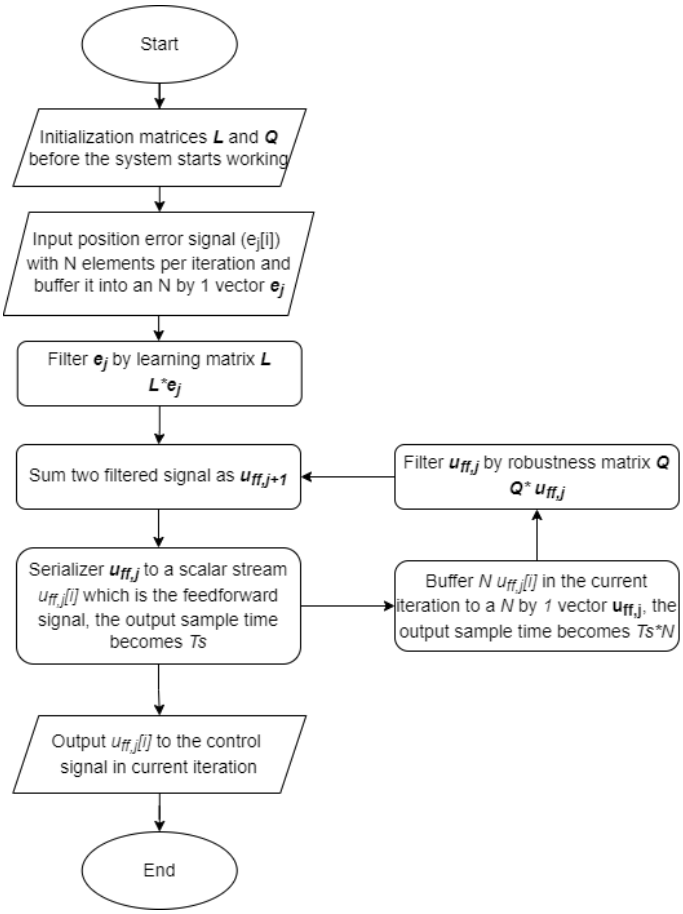

Figure 4.3: Original ILC working flow

Based on Equation 4.12, $\mathbf{L}$ and $\mathbf{Q}$ can be calculated based on the Equation 4.13 and Equation 4.14 [73].

$$\mathbf{L} = [(\mathbf{PS})^T \cdot \mathbf{W_e} \cdot \mathbf{PS} + \mathbf{W_f} + \mathbf{W_{\Delta f}}]^{-1}[(\mathbf{PS})^T \cdot \mathbf{W_e}] \tag{4.13}$$

$$\mathbf{Q} = [(\mathbf{PS})^T \cdot \mathbf{W_e} \cdot \mathbf{PS} + \mathbf{W_f} + \mathbf{W_{\Delta f}}]^{-1}[(\mathbf{PS})^T \cdot \mathbf{W_e} \cdot \mathbf{PS} + \mathbf{W_{\Delta f}}] \tag{4.14}$$

With the knowledge of matrices $\mathbf{L}$ and $\mathbf{Q}$, the ILC algorithm can be simply considered as a sum of two products of matrix multiplications. $u_{ff,j+1}$ is the sum of filtered e_j in the current iteration and the filtered $u_{ff,j}$ in the current iteration. The NOILC update equation is presented in the Equation 4.15.

$$\mathbf{u_{ff,j+1}} = \mathbf{L} \cdot \mathbf{e_j} + \mathbf{Q} \cdot \mathbf{u_{ff,j}} \tag{4.15}$$

The ILC control diagram is presented in Figure 4.2. The two buffers in the diagram is used to converge two one-dimension signals $e_j[i]$ and $u_{ff,j}[i]$ into two N by 1 vectors $\mathbf{e_j}$ and $\mathbf{u_{ff,j}}$. After buffering the position error in the j^{th} iteration into an N by 1 vector, $\mathbf{e_j} \in \mathbb{R}^N$, it is right multiplied by $\mathbf{L}$. At the same time, the feedforward signal in the j^{th} iteration is also buffered to an N by 1 vector, $\mathbf{u_{ff,j}}$ and right multiplied by $\mathbf{Q}$. The sum of these two results is the vector of the feedforward signal, $\mathbf{u_{ff,j+1}} \in \mathbb{R}^N$, which is added to the control signal in the $(j+1)^{th}$ iteration after $\mathbf{u_{ff,j+1}}$ being serialized into the one-dimension signal $u_{ff,j+1}$. The whole workflow is presented in Figure 4.3.

4.3 NOILC with Classical Method

4.3.1 Definition of Matrix Multiplication

The classical matrix multiplication takes $\mathcal{O}(n^3)$ computations [76] to calculate two matrix with $\mathcal{O}(n^2)$ elements. Considering the inputs of the computation to be a m by p matrix $\mathbf{A}$ and a p by n matrix $\mathbf{B}$, the result matrix $\mathbf{C}$ will be an m by n matrix.

$$\mathbf{C} = \mathbf{A} \times \mathbf{B} \tag{4.16}$$

(4.17)

$$\begin{bmatrix} C_{1,1} & C_{1,2} & \cdots & C_{1,n} \\ C_{2,1} & C_{2,2} & \cdots & C_{2\,n} \\ \vdots & \vdots & \ddots & \vdots \\ C_{m,1} & C_{m,2} & \cdots & C_{m,n} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,P} \\ A_{2,1} & A_{2,2} & \cdots & A_{2\,P} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,P} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,n} \\ B_{2,1} & B_{2,2} & \cdots & B_{2\,n} \\ \vdots & \vdots & \ddots & \vdots \\ B_{P,1} & B_{P,2} & \cdots & B_{P,n} \end{bmatrix}$$

Figure 4.4: classical Matrix Multiplication Operation

During the computation, each element in matrix $\mathbf{C}$ is concerned with one row in matrix $\mathbf{A}$ and one column in matrix $\mathbf{B}$ like Figure 4.4 presented. For any element $C_{i,j}$, in the k^{th} row and l^{th} column equals to the sum of dot products of the whole k^{th} row of matrix $\mathbf{A}$ and the whole l^{th} column of matrix $\mathbf{B}$ shown in Equation 4.18. Thus, p multiplications and additions are required to compute one element in the output. In total $p \times m \times n$ multiplications and additions are required to complete a matrix multiplication.

$$C_{k,l} = A_{k,1} \times B_{1,l} + A_{k,2} \times B_{2,l} + \ldots + A_{k,q} \times B_{q,l} + \ldots + A_{k,p} \times B_{p,l}, q \in \{1, 2, \ldots, p\} \tag{4.18}$$

$$= \sum_{q=1}^{p} A_{k,q} \times B_{q,l} \tag{4.19}$$

4.3.2 Matrix Multiplication Computation in Parallel

To ensure sufficient computing time, all calculations are designed to be performed simultaneously by employing multiple processing units. The number of employed processing units equals to the number of multiplications for each iteration. Each processing unit performs the required multiplication operations at the same time. Based on the multiple DSP cores on the FPGA, the FPGA is able to break the matrix multiplication, into separate scalar multiplications. Each DSP core executes one scalar multiplication. Thus, scalar multiplication operations in a matrix multiplication are able to be processed at the same time rather than waited to be processed in series. The computation time could be saved but a lot of DSP cores are employed to achieve the matrix multiplication. Since the number of required DSP cores may exceed hundreds or even thousands where only tiny FPGAs are designed with such a number of DSPs and these FPGAs are usually expensive [4], all multiplication operations can be performed at the same time, but it is costly.

For one matrix multiplication in the ILC system, N^2 multipliers are required. This number increases to a huge scalar with the rising of N. Moreover, ILC operates two matrix multiplications which double the utilization of multipliers. Generally, FPGAs are not designed with so many DSPs, XC7Z100 in the Zynq-7000 family contains the maximum number of DSPs which is 2020 [19]. There exist FPGA boards with thousands of DSPs but those are much more expensive [4]. When the system has a small N, it is possible to implement the NOILC by the method mentioned in this section.

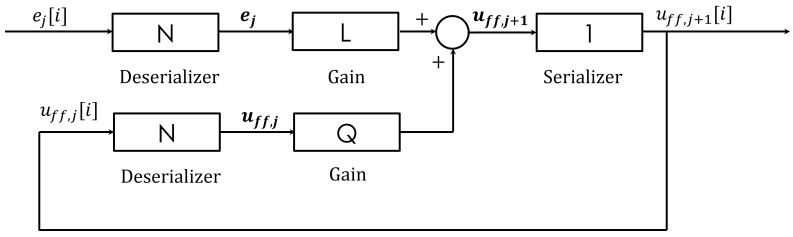

Figure 4.5: Diagram of classical NOILC Implementation

4.3.3 Resource utilization Model

The diagram in Figure 4.5 is a NOILC system constructed with the classical matrix multiplication in parallel. Signal e_j and $u_{ff,j}$ have to be buffered into a vector for computation. Thus, in the buffering step also called deserializing step, at most $2N$ samples need to be saved in the chip. Assuming the bit-width of e_j and $u_{ff,j}$ is bW, $2bW \cdot N$ bits need to be saved in storage blocks. The vector can be either stored in a block RAM or in distributed memory in an FPGA directly. Commonly saving a large vector in block RAMs would be an advisable choice since distributed memory in an FPGA is restricted. However, considering that all the elements have to be computed at the same time, it is hard to send hundreds of or even thousands of elements from block RAMs to operational blocks in a relatively short period of time. Thus, these elements would be better stored in the distributed RAMs in the FPGA. The deserializing step would be achieved by adopting flip-flops (FFs).

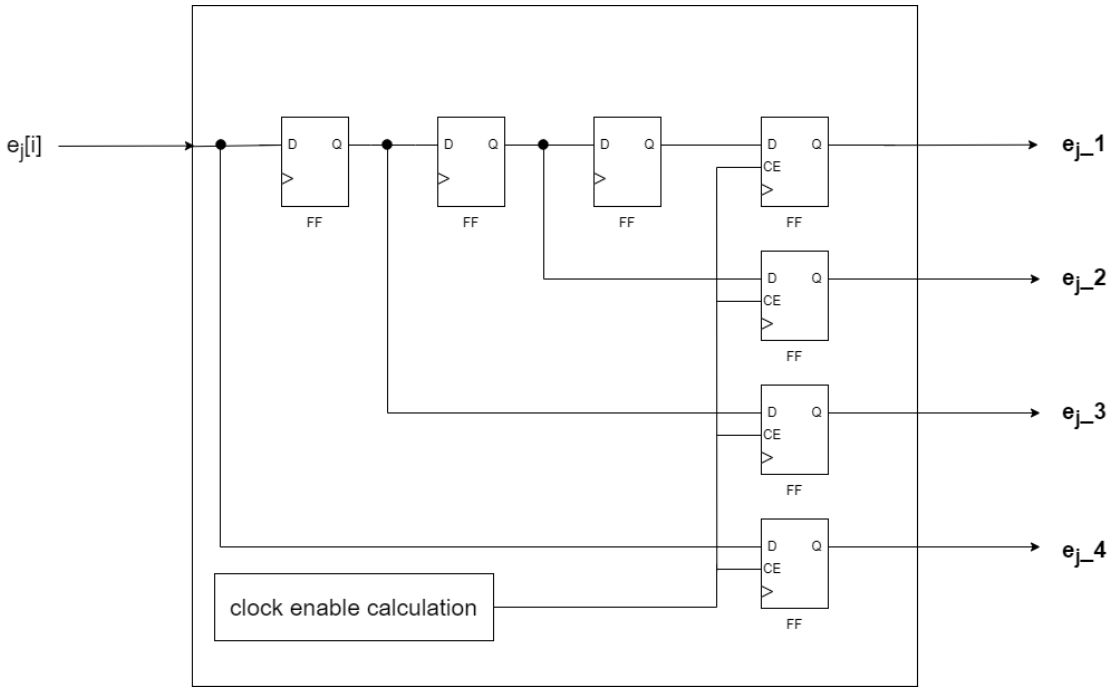

Figure 4.6: The functional diagram of a deserializer when N = 4, bit-width = 1

The deserializer constructed in this step must be able to buffer a scalar stream into a N by 1 vector and output all elements in the vector after receiving the final element. Therefore, as the bit-width is bW, there should exist $N \cdot bW$ of data FFs connected with a clock enable signal which guarantees elements in the vector output at the same time and additional $N-1$ FFs help to buffer

the earlier elements like Figure 4.6 shown. Thus, $bW(2N-1)$ FFs are required to achieve the buffering function for one matrix multiplication in the ILC algorithm. To realize the number of input elements and prevent the deserializer outputting until enough elements have been buffered, it needs an adder to count the number of elements N and a comparer to compare the result of the adder with N. $\lceil log_2(N) \rceil$ FFs and $\lceil log_2(N) \rceil$ LUTs constitute an adder and $\lceil log_2(N) \rceil$ FFs constitute a comparer. The $\lceil x \rceil$ rounds the element x to the nearest integer which is greater or equal to itself. The clock enable signal for the FFs in the deserializer mentioned before should not be an external signal. It should be calculated based on the input sample period and the size of buffered vector. An adder and a comparer are required to calculate the clock enable signal and an additional LUT might be required to control the level of the clock enable signal applied to the FFs. A deserializer does not need any digital signal processing (DSP) blocks and required the total number of FFs, LUTs are:

$$\begin{cases} U_{LUT} = 2 \times (1 + 4 \times \lceil log_2(N) \rceil) & (4.20) \\ U_{FF} = 2 \times (bW \times (2 \times N - 1) + 2 \times \lceil log_2(N) \rceil) & (4.21) \\ U_{DSP} = 0. & (4.22) \end{cases}$$

There are more efficient deserializer designs. The two sets of adders and comparers, for example, perform the same purpose. A deserializer can be designed with just one adder and one comparer. The suggested constitution is nearly the worst case scenario, utilizing as much resources as possible. As a result, if the actual resource on the platform is more than the calculated resource, the design will not fail due to a lack of resources. There do exist the designs of the deserializer which consumes more resources like the design in Figure 4.7 where the signal output from the first FF in the first line is same with the output of the FF in the fourth line. The six FFs output of the first line are repeating the same behaviours of the FFs in the first line. The weakness of the design like this is able to be optimized by the synthesis tool. Thus, the design in Figure 4.7 is not considered to be the worst design and the resource utilization is not predicted based on it.

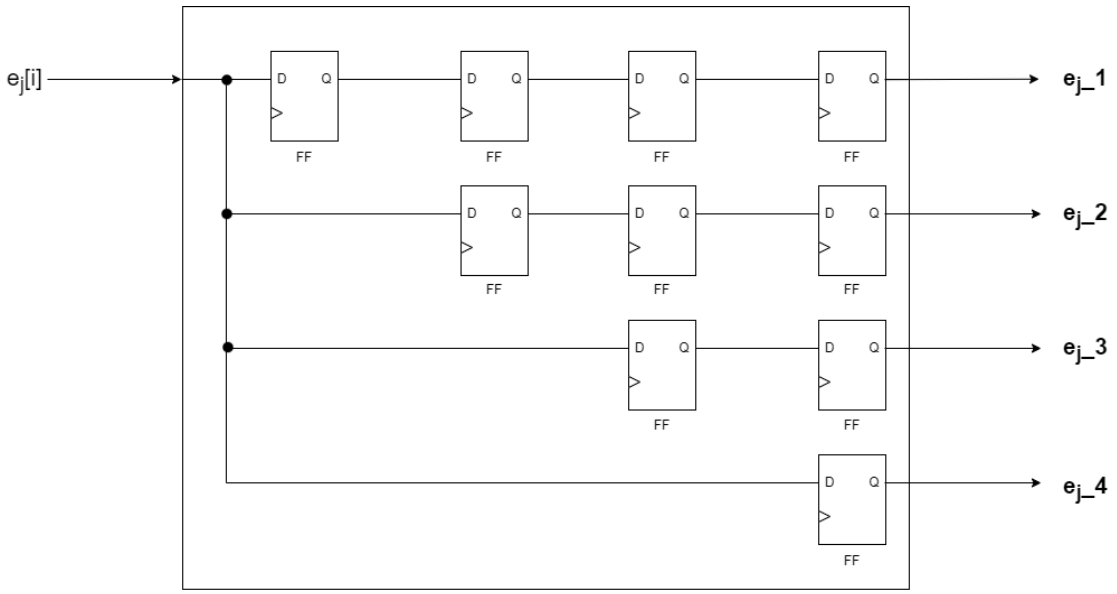

Figure 4.7: A terrible design of a deserializer when N = 4, bit-width = 1

To guarantee that two matrix-vector multiplications in ILC output their results simultaneously, $2 \times N \times N$ multipliers are required to achieve this parallel execution since there exist two vector-matrix multiplications in 4.15. There exist two buffered vectors, two matrices which can be either saved in block RAM or distributed RAM in the FPGA. Usually, such a large amount of data should not be stored in distributed memory but in the worst situation, MATLAB may design to

store these elements in distributed memory because matrices are not changed by time and all of the elements join the multiplications in every iteration. To consider the worst situation, two matrices are assumed to be saved in distributed memory built up by LUTs. For $2 \times N \times N$ elements, $bW \times 2 \times N \times N$ LUTs are cost, and each multiplier needs one DSP to complete operation. The resources required for the multiplication part are:

$$\begin{cases} U_{LUT} = bW \times 2 \times N \times N & (4.23) \\ U_{FF} = 0 & (4.24) \\ U_{DSP} = 2 \times N \times N. & (4.25) \end{cases}$$

Table 4.1: The resource utilization assumption for the classical NOILC implementation

	LUTs	FFs	DSPs
Deserializer (2)	$2(1+4\lceil log_2(N) \rceil)$	$2(bW(2N-1)+2\lceil log_2(N) \rceil)$	0
Matrix multiplication (2)	$2 \times bW \times N^2$	0	$2N^2$
Total	$2(bW \times N^2+1+4\lceil log_2(N) \rceil)$	$2(bW(2N-1)+2\lceil log_2(N) \rceil)$	$2N^2$

According to the above assumptions, the total resource utilization before serializing is in the Table 4.1. According to the table, it is clear that the utilization of DSP is quadratically increasing with N. When the FPGA platform does not have enough physical DSPs, to achieve the function of the design, MATLAB might allowed one DSP to process more than one multiplications in series. Because the input data cannot be processed directly, part of them have to be stored and wait to be processed until the physical DSPs finish the previous multiplications. So does the output of DSPs must wait for all products computed. Thus, more data has to be stored for waiting, more FFs are employed to achieve the storage. Moreover, more selections are made to decide which multiplication or addition is going to be performed and more LUTs are utilized for the selection. Thus, when N rises to a large number resulting insufficient DSPs, this process leads to the surge in the utilization of LUTs and FFs. The actual resource utilization becomes unpredictable. To achieve an ILC system running on cost-optimized devices or mid-range devices, the utilization of DSPs has to be decreased.

4.4 NOILC with Improved Method

The number of elements of position error, N, in factories ranges from hundreds to thousands. In the last section, it is proved that when more than one multiplications have to share one physical DSP to achieve computations, with the increase of N, the required LUTs and FFs may increase to a huge number. This number is probably larger than the number of FFs and LUTs the board actually has leading to the system cannot be executed on the selected FPGA platform. Therefore, in order to ensure that the algorithm can operate even with a large N, the resource utilisation of the algorithm should be improved.

4.4.1 Matrix Multiplication by Column Combination Method

Because the dramatic increase in LUTs and FFs occurs when there are not enough DSPs, for the current algorithm, the core of resource utilization reduction should be reducing the DSP usage. In the previous approach, with the increase of N, the usage of the DSPs is quadratic growth (DSPs=$2N^2$). Thus, it is better to limit the growth of DSP utilization to linear growth or even slower.

Block matrix multiplication mentioned in Section 3.2 decreases the actual resource utilization on the FPGA. Considering that in NOILC, only when enough of elements are entered, the vector

can be buffered by these elements. The N-size-vector buffering process requires $T_s(N-1)$ to complete. Block matrix multiplication still needs to buffer a smaller vector spending time in an iteration. The best solution is that the system would not waste time in buffering.

If elements from the input position error and feedforward signals in the current iteration, $e_j[i]$ and $u_{ff,j}[i]$ can be processed in time rather than waiting until all samples are buffered into two vectors, the computation time would not be as narrow. Thus, the allowable computation time for all multiplications increases from T_s to $N \times T_s$, the length of one iteration.

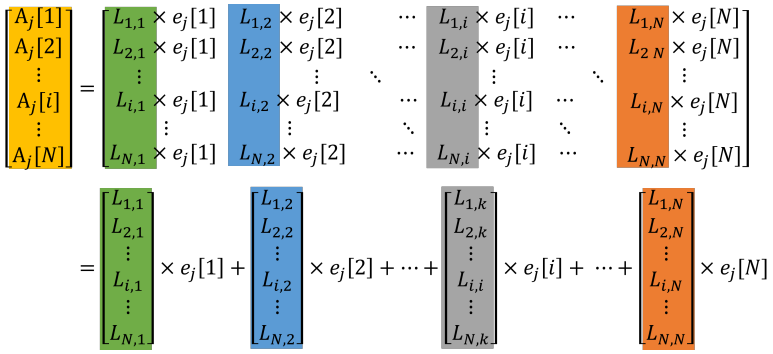

Figure 4.8: Right vector-matrix multiplication

For one matrix multiplication, $\mathbf{L}$ multiplying with the position error $\mathbf{e_j}$, in ILC system, we assume its result as an N by 1 vector $\mathbf{A_j}$. There is an N by 1 vector on the right side of multiplication, visualizing from Figure 4.8, the i^{th} element of $\mathbf{e_j}$ is only operated with the i^{th} column of $\mathbf{L}$. Thus, the result vector $\mathbf{A_j}$ can be represented by Equation 4.26.

$$\mathbf{A_j} = \sum_{i=1}^{N} (e_j[i] \times \mathbf{L_{c,i}}) \tag{4.26}$$

where $\mathbf{L_{c,i}}$ is the i^{th} column of matrix $\mathbf{L}$.

$$\mathbf{L} = \begin{bmatrix} \mathbf{L_{c,1}} & \mathbf{L_{c,2}} & \dots & \mathbf{L_{c,i}} & \dots & \mathbf{L_{c,N}} \end{bmatrix} \tag{4.27}$$

$$\mathbf{L_{c,i}} = \begin{bmatrix} L_{1,i} \\ L_{2,i} \\ \vdots \\ L_{i,i} \\ \vdots \\ L_{N,i} \end{bmatrix}, \mathbf{L_{c,i}} \in \mathbb{R}^N \tag{4.28}$$

4.4.2 NOILC Implementation by Improved Method

According to Equation 4.26, if at the time when $e_j[i]$ enters the ILC system, the corresponding vector $L_{c,i}$ is selected to operate with $e_j[i]$ while the product is buffered for later accumulation. It works the same on the Q filter part. Because the matrices $\mathbf{L}$ and $\mathbf{Q}$ are calculated off-line, the values are available to be stored before the system starts working. Thus, elements from the position error and feedforward signal do not need to be buffered into two vectors and the computations do not need to wait until the last T_s in each iteration.

When the column combination is performed, the number of multipliers required grows linearly with the number of items in one iteration (N). N multipliers are utilized to compute $e_j[i] \times \mathbf{L_{c,i}}$.

To decrease the number of multipliers, the multiplication between $e_j[i]$ and $\mathbf{L_{c,i}}$ is arranged in series by decreasing the sample time of $e_j[i]$ from T_s to T_s/N, multiplying $e_j[i]$ with N elements in $\mathbf{L_{c,i}}$ serially. The allowable computation time for each matrix multiplication becomes T_s/N. When the sampling frequency is 10 kHz and the number of elements N is 1000 samples per iteration, the allowable computation time for one multiplication should be 100 ns. Therefore, the clock frequency must be larger than 10 MHz which is supported by almost all FPGA boards. In this case, as long as the computation time for each multiplication operation is within the limitation of selected FPGAs, only one multiplier is required for one matrix multiplication. The DSP utilization is restricted to a very small range which would not increase with N. The number of DSPs on a platform would not limit the implementation of NOILC on FPGA.

The buffered product of each column should be added together and output as the feedforward signal after being serialized. If the addition process does not start until the last product result is calculated in one iteration, $N \times N$ elements need to be stored. Because the signal $e_j[i]$ does not need to be buffered, the multiplications between $e_j[i]$ and element in $\mathbf{L_{c,i}}$ are processed immediately after receiving $e_j[i]$. The addition operation in matrix multiplication does not need to wait for the completeness of the multiplication operation. Thus, the accumulation and multiplication can be processed synchronously.

$$\mathbf{u_{ff,j+1}} = \mathbf{L} \cdot \mathbf{e_j} + \mathbf{Q} \cdot \mathbf{u_{ff,j}} \tag{4.29}$$

$$\begin{bmatrix} u_{ff,j+1}[1] \\ u_{ff,j+1}[2] \\ \vdots \\ u_{ff,j+1}[i] \\ \vdots \\ u_{ff,j+1}[N] \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^{N}(e_j[k] \times L_{1,k}) \\ \sum_{k=1}^{N}(e_j[k] \times L_{2,k}) \\ \vdots \\ \sum_{k=1}^{N}(e_j[k] \times L_{i,k}) \\ \vdots \\ \sum_{k=1}^{N}(e_j[k] \times L_{N,k}) \end{bmatrix} + \begin{bmatrix} \sum_{k=1}^{N}(u_{ff,j}[k] \times Q_{1,k}) \\ \sum_{k=1}^{N}(u_{ff,j}[k] \times Q_{2,k}) \\ \vdots \\ \sum_{k=1}^{N}(u_{ff,j}[k] \times Q_{i,k}) \\ \vdots \\ \sum_{k=1}^{N}(u_{ff,j}[k] \times Q_{N,k}) \end{bmatrix} \tag{4.30}$$

$$\begin{bmatrix} u_{ff,j+1}[1] \\ u_{ff,j+1}[2] \\ \vdots \\ u_{ff,j+1}[i] \\ \vdots \\ u_{ff,j+1}[N] \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^{N}(e_j[k] \times L_{1,k} + u_{ff,j}[k] \times Q_{1,k}) \\ \sum_{k=1}^{N}(e_j[k] \times L_{2,k} + u_{ff,j}[k] \times Q_{2,k}) \\ \vdots \\ \sum_{k=1}^{N}(e_j[k] \times L_{i,k} + u_{ff,j}[k] \times Q_{i,k}) \\ \vdots \\ \sum_{k=1}^{N}(e_j[k] \times L_{N,k} + u_{ff,j}[k] \times Q_{N,k}) \end{bmatrix} \tag{4.31}$$

Because the products from one column do not interact with each other and are computed sequentially, they can be buffered into a N by 1 vector and added with other buffered products. After storing the first column's product vector into distributed memory, the vector is allowed to be updated, becoming the product vector of the next column in the following sample time by reusing the same storage. The updated vector is then added to the stored vector, and the sum of them is saved in the same memory location as the stored vector. By doing so, the accumulation is completed after N repetitions of the same process. The accumulation is the final result of this matrix multiplication at the last sample time in each iteration, after the last column products being added. N adders are required for accumulation, and the permitted calculation time for one matrix multiplication in NOILC becomes $N \cdot T_s$; additionally, only $2N$ elements must be arranged to store in the chip.

Based on the Equations (4.29 - 4.31) it is clear that the k^{th} element of $u_{ff,j+1}$ equals to the sum of N times addition ($u_{ff,j+1}[i] = \sum_{k=1}^{N} e_j[k] \cdot L_{i,k} + u_{ff,j}[k] \cdot Q_{i,k}$). By breaking the multiplication and addition in matrix multiplication into two parts, this method provides

an opportunity to perform the multiplication of $e_j[i+2]$ and elements $\mathbf{L_{c,i+2}}$ and addition $e_j[i+1] \cdot \mathbf{L_{c,i+1}} + \sum_{k=1}^{i}(e_j[i] \cdot \mathbf{L_{c,k}})$ at the same time. According to Equation 4.31, adding the two products before buffering them into vector and then buffering them into a vector for accumulation has the same result with adding the two accumulated vectors. The benefit of adding two products before buffering is that N adders for one matrix multiplication would be saved and the storage for the $2N$ elements would not need to be arranged anymore.

The workflow of the ILC system implemented by the column combination method is shown in Figure 4.9. Since the matrices $\mathbf{L}$ and $\mathbf{Q}$ are known and would not be changed anymore for a specific system, these two matrices can be preloaded into block RAMs, otherwise, saving these two matrices in distributed memory would waste a lot of FFs, even use up all FFs when N is large. After the system start, elements of e_j enter the ILC system with sample period T_s. Then the sample period of e_j reduces to T_s/N to multiply with the N elements from the corresponding column of matrix $\mathbf{L}$ sequentially. At the same time, the sample period of the feedforward signal $u_{ff,j}$ also reduces from T_s to T_s/N and multiplies with the elements in the corresponding column of the matrix $\mathbf{Q}$. The products from $e_j[i]$ and N elements of $\mathbf{L_{c,i}}$ are added with the products of $u_{ff,j}[i]$ and N elements of $\mathbf{Q_{c,i}}$. These N additions will be buffered into a N by 1 vector for accumulation. At the start of each iteration, the adding factor should be initialized to zero. After N addition operations in the accumulation process, the result is the vector of the feedforward signal which needs to be serialized and output with sample time T_s.

4.4.3 Resource Utilization Model

By using the column combination method described in Section 4.4.2, only two DSPs are utilized for two matrix multiplications regardless the value of N. The utilization of DSPs would not increase with the growth of N. Thus, for a system with a large N, the number of DSPs on the FPGA is also sufficient. The addition of two products needs one adder. Considering the bit-width, bW LUTs are required for the adder. To buffer the sum of two products into a N by 1 vector, a deserializer is necessary which requires $1 + 4\lceil log_2(N) \rceil$ LUTs and $bW(2N-1) + 2\lceil log_2(N) \rceil$ FFs as mentioned in Section 4.3.3. The elements of matrices $\mathbf{L}$ and $\mathbf{Q}$ can be stored in block RAMs because the elements are used sequentially.

Only N adders are required for accumulating the sum of products. The total number of LUTs required is $bW \times N$. To achieve the loop of accumulation, the system needs a switch, a counter and a constant 0. The constant 0 is to initialize the storage at the start of each iteration. An another switch is required to identify whether output the final result at the end of each iteration. To count for number N, it needs $2 \times \lceil log_2(N) \rceil$ LUTs and $\lceil log_2(N) \rceil$ FFs as described in Section 4.3.3. One LUT is necessary for constant 0. The inputs and outputs of two switches are both N by 1 vectors, therefore each switch needs to do N selections which requires $bW \times N$ LUTs.

Table 4.2: The resource utilization assumption for the improved NOILC implementation

	LUTs	FFs	DSPs
Deserializer	$1 + 4\lceil log_2(N) \rceil$	$bW(2N-1) + 2\lceil log_2(N) \rceil$	0
Multiplier(2)	0	0	2
Adder of products	bW	0	0
Counter	$2\lceil log_2(N) \rceil$	$\lceil log_2(N) \rceil$	0
Constant 0	1	0	0
Adders for accumulation	$bW \cdot N$	0	0
Switch (2)	$2bW \cdot N$	0	0
Total	$3bW \cdot N + 6\lceil log_2(N) \rceil + bW + 2$	$bW(2N-1) + 3\lceil log_2(N) \rceil$	2

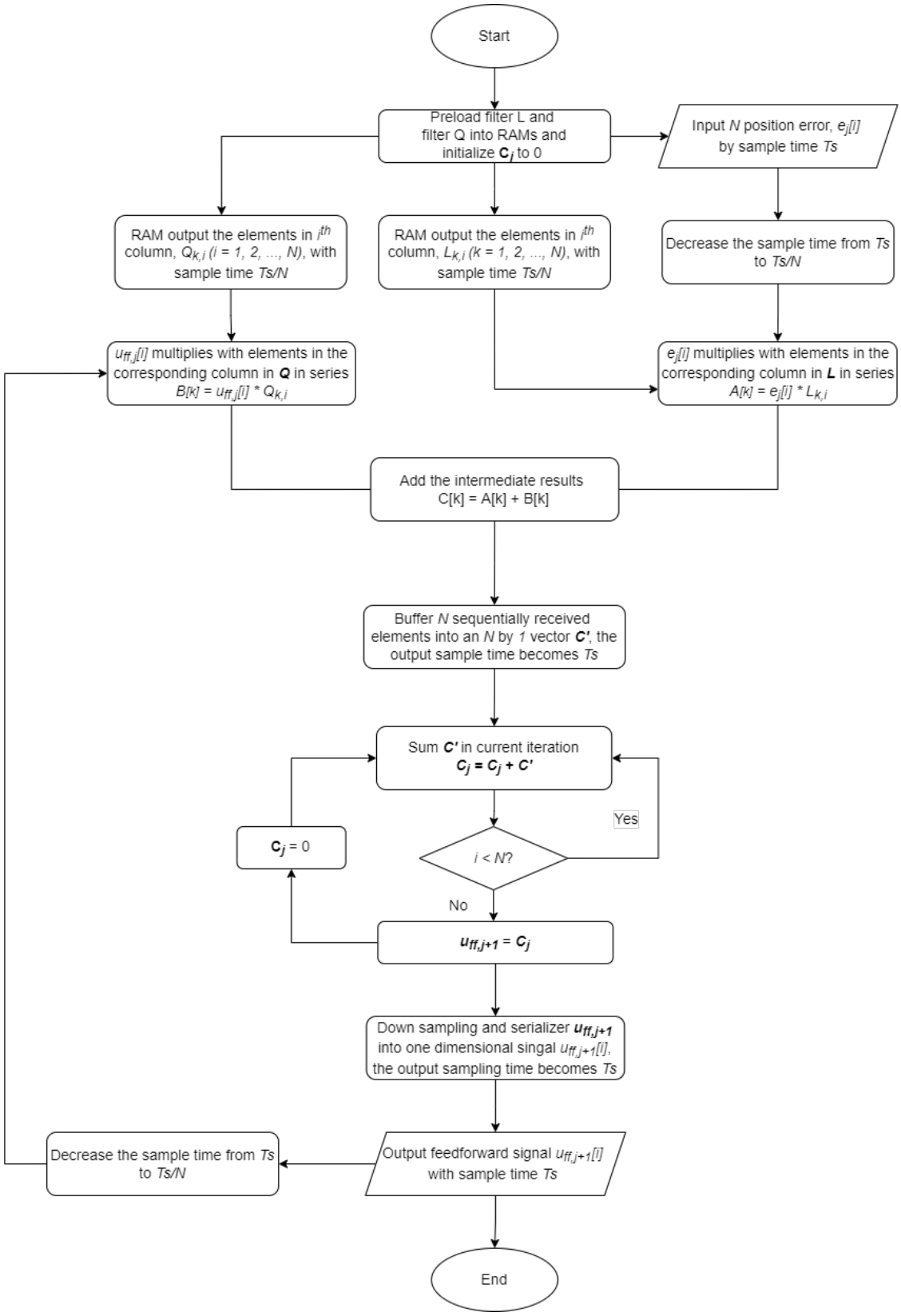

Figure 4.9: Improved ILC working flow by using Column Combination Method

Table 4.3: Comparison of resource usage of the classical and the improved methods

	LUTs	FFs	DSPs
Classical method	$2(bW \times N^2 + 1 + 4\lceil log_2(N) \rceil)$	$2(bW(2N-1) + 2\lceil log_2(N) \rceil)$	$2N^2$
Improved method	$3bW \cdot N + 6\lceil log_2(N) \rceil + bW + 2$	$bW(2N-1) + 3\lceil log_2(N) \rceil$	2

The total resource cost for the improved method is in Table 4.2. Compared with the classical implementation method in table 4.1, the rise of LUTs becomes a linear with the development of N, and the use of DSPs is a constant. In this project, only 2 DSPs are utilized. The usage of FFs linear increases with the growth of N. Since the computation time for multiplication is related to the number of elements in one iteration, for a system with a huge N, the computation time for multiplication may decrease to 10^{-10}s which is almost impossible to be processed. In this case, more multipliers should be utilized to do multiplications in parallel, reducing calculation time. Another disadvantage of this design is that the sample frequency of the output feedforward signal may not be f_s. The sample frequency of the output signal $u_{ff,j+1}$ is possible to be faster than f_s because the sampling frequency of the signals e_j and $u_{ff,j}$ in the previous step is raised in order to achieve multiplication. Before adding the feedforward signal with the feedback control signal, the sampling frequency of $u_{ff,j+1}$ must be decreased to f_s by downsampling.

Chapter 5

Experimental Setup

An FPGA is required to process the projects while constructing a NOILC system on an FPGA platform. To simulate execution, control systems should be constructed in MATLAB Simulink. When the simulation results show that the systems converge and that the feedforward signals created by ILC help in reducing error and speeding up convergence, the HDL projects of ILC algorithms are generated via the FPAG programming process provided by MATLAB. Finally, the project may be executed on the selected FPGA board, and the actual resource utilization can be checked in the project files.

5.1 Platform

Not all the FPGA devices on the market are supported by the FPGA programming of MATLAB. Zedboard is one of the supported hardware devices, meanwhile, meanwhile, at $475, Zedboaed is not prohibitively pricey [18]. Zedboards can process designs based on Linux, Windows, and other real-time operating systems [18]. The Zynq chip can be divided into two parts, processing system (PS) and programmable logic (PL) [19]. PS is a traditional processor equipped with a dual-core ARM Cortex-A9 processor, memory, timer, various communication interfaces for external devices, etc. Zedboard is designed with Xilinx XC7Z020 as a programmable logic device which contains 53200 LUTs, 106400 FFs, 220 DSP slices and 4.9Mb total block RAM [19]. PS is interconnected with PL via internal high-speed buses.

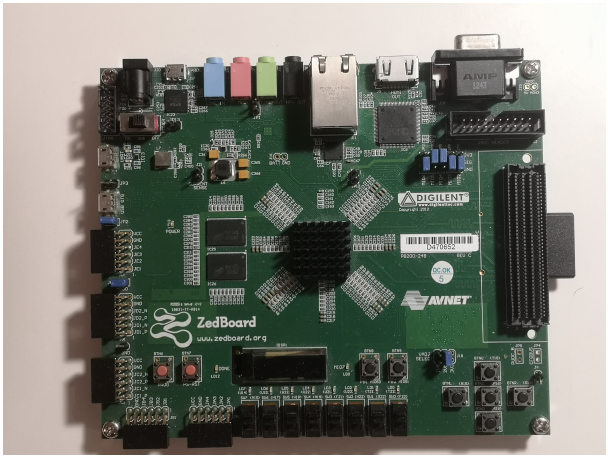

Figure 5.1: The chosen platform: Zedboard

5.2 NOILC with Classical Matrix Multiplication

5.2.1 NOILC Implementation

A traditional feedback control system is made up of two major components: a controller and a plant. This control system's objective is to ensure that the actuator controlled by the motor goes to the specified position. As a result, the plant in this project is a continuous actuator directed by a motor. The signal θ_j, on the other hand, is the measured encoder position in the discrete-time domain, reading data based on the sampling frequency. Because MATLAB is more likely to report errors when developing HDL projects for hybrid systems than when developing projects for exclusively discrete systems. Thus, despite the fact that the plant should be designed in the continuous-time domain, the simulated plant is constructed in the discrete-time domain. The feedback control system is constructed based on a given simulated model with a sampling frequency 16 kHz.

In order to buffer the one-dimensional input signal $e_j[i]$ and output signal $u_{ff,j}[i]$ into two N by 1 vectors, two buffers are required. In Simulink, the component *Buffer* is able to achieve this function [5]. The component *Gain* [12] is able to perform multiplication with provided gain factors which can be either scalars, vectors or matrices. These gain factors are kept in distributed memory and are connected to the same clock signals as the multipliers. Thus, the input signals are multiplied by these factors as long as the multiplications are processed. These gain factors are accompanied by the multipliers to guarantee that as long as the input signal comes, multiplications would be processed. Two *Gain* components achieve the vector-matrix multiplications by providing matrices L and Q as gain factors. After adding the two products, the final vector $\mathbf{u_{ff,j}}$ should be converted into a one-dimensional signal again, which is also achievable by *Buffer*. However, *Buffer* does not support the HDL code generation in MATLAB. The substitutes for buffers are *Deserializer1D* [9] and *Serializer1D* [14] who convert a one-dimensional signal into vectors and vectors into lower dimension signal respectively.

5.2.2 Number of Samples Reduction

The reference signal contains 650 samples at 16 kHz sampling frequency in each iteration (Figure 5.2 (a)), which cannot be implemented on the chosen Zedboard because the number of expected multipliers (422500) is much greater than the number of DSPs a Zedboard has (220). To implement the current NOILC algorithm on the chip, the number of samples has to be reduced without influencing the convergence of the system.

Currently, after reaching the predicted location from the initial position, the trace of the reference position remains in the current state for a short time before returning to the initial position to await movement in the next iteration. However, ILC cannot identify whether the command position is back to the initial position or not since once the control system converges, the signal e_j reduces to almost zero and the input signal to ILC is the position error. Therefore, the reference position does not have to return to the initial position but increases directly based on the convergence state forming as a stair-step signal. The number of required samples to achieve convergence becomes only half of the original one as 325 as figure (b) in Figure 5.2.

5.2.3 Word Width

The NOILC is implemented on the FPGA while the feedback control part is programmed on the ARM. The communication between these two parts is based on the AXI4 interfaces arranged by Simulink. Because the datatype is not specified by the designer during the simulation, MATLAB utilises double precision as the datatype. To transmit data between ARM and FPGA, the signal e_j and $u_{ff,j}$ would be better to be converted to a signed fixed-point data type. With the help of

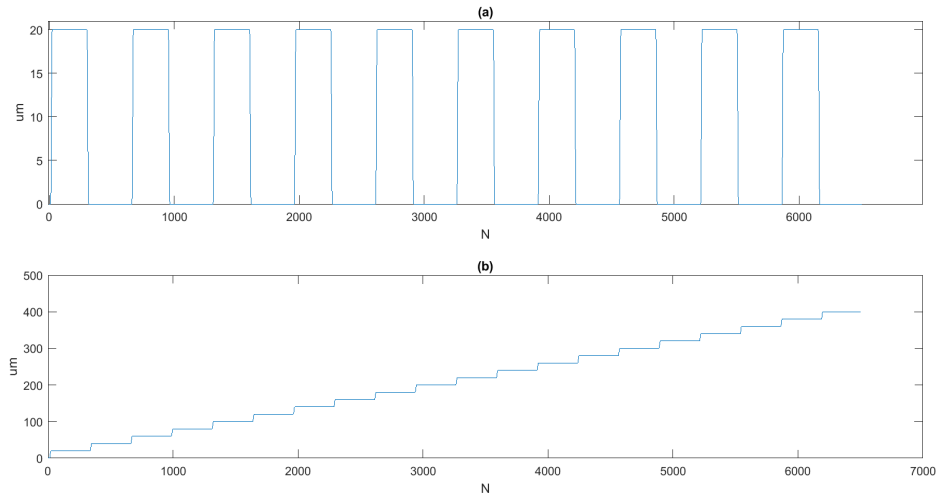

Figure 5.2: Two reference position signals with different lengths

the feedforward signal generated by NOILC, the error starts to reduce and gradually approaches a value of 0 from the second iteration. Thus, the maximum difference between the reference position signal and encoder measurement signal should be in the first iteration. In contrast, the signal $u_{ff,j}$ remains at 0 in the first iteration and it has the computed output from the second iteration. The values of $u_{ff,j}[i]$ remain stable after the system becomes convergent which may take several iterations. Based on the maximum values and the minimum values of signal e_j and $u_{ff,j}$ measured in the simulation, the shortest word length to ensure the system working correctly can be confirmed. But overflow easily happens because the result from the hardware cannot be totally the same with the simulation result. In this project, to preserve the storage on FPGA, the shortest world width is chosen. The minimum and maximum values of e_j and u_ff in 10 iterations are:

- max e_j: 394.7204
- min e_j: -368.8130
- max $u_{ff,j}$: 1223.8462
- min $u_{ff,j}$: -1560.7651.

The data type must be signed and it needs at least 11 bits for the integer part ($log_2(1560) = 10.60733031$). After measuring the fractional part, the word width of signals is 24 bits with 12 bits of fraction, 11 bits of integer and 1 signed bit. According to Figure 5.3, the signal in fixed point is almost overlapping with the signal in double precision. The maximum difference in 10 iterations is around 1 um which is acceptable. Thus, the NOILC part is implemented with 24 bits signed fixed-point format.

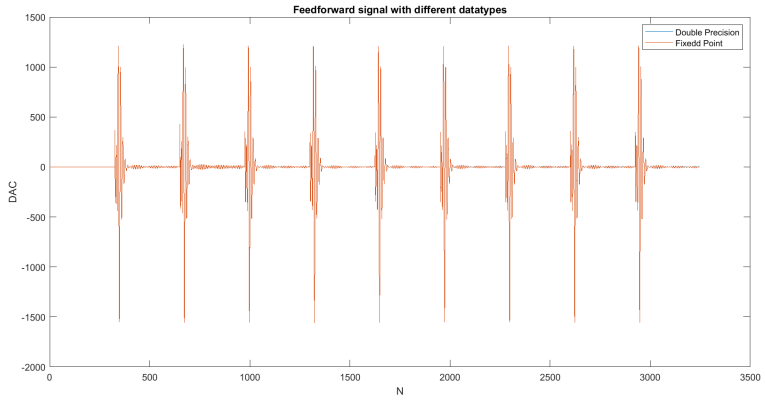

Figure 5.3: The feedforward signal $u_{ff,j}$ with different data types

5.3 NOILC with Improved Matrix Multiplication

To construct the improved model described in Section 4.4.2 in MATLAB Simulink, two RAMs are required to store the elements of matrices **L** and **Q**. Two *Product* [13] components are utilized as multipliers. An adder is followed by the multipliers outputting the sum of products. A *Deserializer1D* described in Section 5.2.1 is utilized as a buffer to convert the one-dimensional sum to a N by 1 vector. Two *Switch* [15] components, an *Adder* [3], a *UnitDelay* [16] for storing the sum of all computed column products, a *Counter* [8] and a *Constant* 0 [7] constitute a for loop for the accumulation of all the column products in one iteration. The output of the loop is the final result $\mathbf{u_{ff,j}}$. The loop initializes the storage as zero at the start of each iteration, making sure that the values in the previous iteration would not affect the calculation in the current iteration. The vector $\mathbf{u_{ff,j}}$ is converted into one-dimensional signal $u_{ff,j}$ by the *Serilizer1D* after downsampling. The signal $u_{ff,j}$ is added to the control signal before the plant and multiplied with elements in Q in the next iteration. It is worth noting that MATLAB Simulink does not allow the interaction of two signals with different sample rates, such as addition and multiplication, otherwise, MATLAB reports errors. If all of these signals have a relationship and their sample rates vary, the sampling rates in Simulink must be altered until they are constant. Thus, two upsampling components and one downsampling component are used to ensure that the output signal has the same sampling frequency as the input signal.

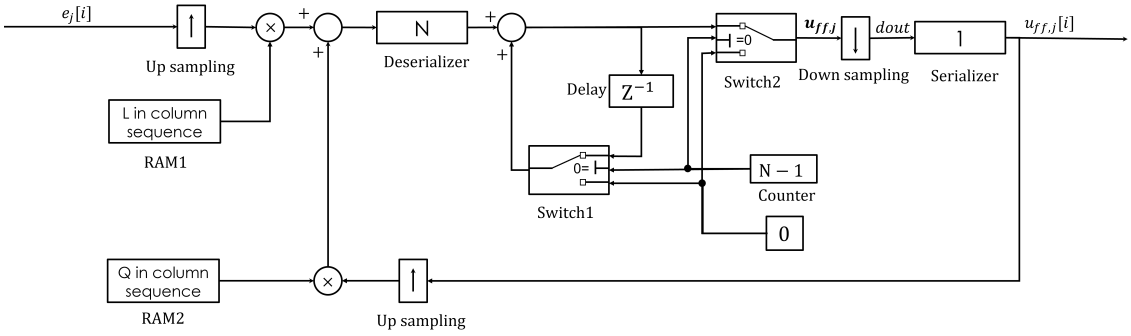

Figure 5.4: Diagram of NOILC implementation with improved method

5.4 Matrix Multiplication Utilization Test

The implementation of NOILC is based on two vector-matrix multiplications with the exact same method and processes. NOILC's total resource utilisation is the double of one signal vector-matrix multiplication with the same size matrix and vector plus an additional adder. Therefore, it is sufficient to analyse only one vector-matrix multiplication. The *Downsampling* component reports a lot of errors while creating HDL project via MATLAB. Nevertheless, the downsampling function is not hard to be achieved by hardware language. For example, the hardware interface of the Simulink, which is an interface allowing models in Simulink to communicate with the design in the hardware, only supports single-rate blocks for concurrent execution while *Downsampling* is a multi-rate block.

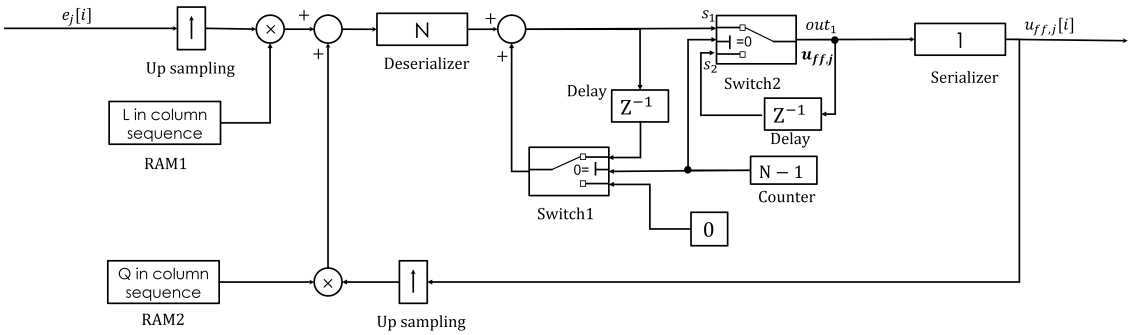

Figure 5.5: The design of improved method without *Downsampling*

By connecting the second input, s_2, of the *Switch2* in Figure 5.5 to output of itself, out_1, with one sample-time delay in between, *Switch2* is able to repeating output the vector $u_{ff,j}$ for $N \cdot Ts$ duration. Thus, out_1 remains $u_{ff,j}$ until $u_{ff,j+1}$ is computed. The signal output from out_1 in Figure 5.5 and the signal output from the *Downsampling*, *dout*, in Figure 5.4 have the same numerical value but the out_1 has higher sampling rate than the *dout*. For the *Serializer* in Simulink, the sample time of the input signal has to be N times larger than the output signal when converting an N by 1 vector to a one-dimension signal [14]. Thus, even though the out_1 has the same value of *dout*, the sample rate of the out_1 has to be decrease. The downsampling process, as well as the block *Downsampling*, cannot be avoided in the Simulink design. But the downsampling design in the HDL design would not be a challenge. It could be achieved by adding FFs, which are connected with a lower rate clock enable signal, connected with the out_1. Therefore, to avoid the negative influence of *Downsampling*, this component would not be utilized for analysis. Due to the influence of *Downsampling*, *Serializer* which follows the *Downsampling* would not be utilized for analysis. Thus, in the classical model analysis, the *Serializer* would not be considered either. The analysis of the classical model only involves a *Deserializer* for buffering and a *Gain* for multiplication.

Because for the classical design only one matrix multiplication is analyzed, only one matrix multiplication is analyzed with the improved model. The entire NOILC only requires one more multiplier and one more adder compared with the one matrix multiplication test. Because of the problems caused by *Downsampling* component in Simulink, *Downsampling* and *Serializer* would not be considered for the the improved model either. The inputs of the improved model are two one-dimension signals and the output is an N by 1 vector. In addition, it outputs the N by 1 vector only in the last T_s of an iteration and outputs 0 in the rest of the same iteration. The improved model contains a multiplier, a *Deserializer*, an adder, a *UnitDelay*, two switches, a counter and a constant 0.

The bit-width (bW) for resource utilization testing is set as 12 bits unsigned fixed-point without a fraction part for the matrix multiplication test. The vectors and matrices for their resource utilization analysis are consisted of randomly generated 12-bit integers. The reason is that there exist a lot of zeros in matrix L and Q. MATLAB may not consider the multiplication with these zeros as operations but constants 0. The tested recourse utilization cannot accurately represents the growth trends of the LUTs, FFs and DSPs. Thus, randomly generated 12-bit unsigned fixed-point data are selected for resource analysis in one matrix multiplication.

Considering that xc7z020clg484-1 Artix-7 FPGA only contains 220 DSPs, when N is larger than 14 ($15^2 = 225$), the number of expected DSPs for the classical matrix multiplication becomes larger than the maximum value it contains. The chip for analysing is changed to xcku5p-sfvb784-1LV-i which has 1824 DSPs allowing the value of N increases to at least 42 ($\sqrt{1824} = 42.7083$).

5.5 HDL Properties Setting Up

To determine the details of HDL code generation and improve performance on time, power, and area, MATLAB gives a lot of distinct properties for various components and designed subsystems. For RAMs in Simulink, for example, *RAMDirective* in HDL properties permits the designer to specify whether the RAM is a block RAM or a distributed RAM. The two chosen properties in this project are *Adaptive Pipelining* and *Clock-rate Pipelining*.

Adaptive Pipelining This property is able to combine certain blocks to reduce the area on the FPGA chip. For components specified by MATLAB, *Adaptive Pipelining* rearranges their designs and deployments providing the opportunity for resource sharing and area optimization [2]. This property is usually utilized with *ShareFactor* which tries to share the *Num* operation with 1 physical resource on FPGA where *Num* is a designed integer number. For example, for a *Gain* component which multiplies a signal with a scalar gain factor, the generated project in register transfer level with selecting *Adaptive Pipelining* or not are presented in Figure 5.6 and Figure 5.7. When the *Adaptive Pipelining* is selected the *Gain* is separated into two parts, a constant gain factor and a multiplier allow this *Gain* to share the physical multiplier with other multiplication if the designer allows the sharing. However, when *Adaptive Pipelining* is set as off, the component *Gain* is considered as a single component which refuses to share any resources with other components. The drawback of *Adaptive Pipelining* is also noticeable where registers are applied to input and output ports of the components which may cause additional delay and more FFs utilization. Thus, *Adaptive Pipelining* does not always optimize on area and timing, sometimes, it may make the situation worse. For example, if there exist only one *Gain* and some selection components in the design, *Adaptive Pipelining* cannot help to share the physical multiplier with other multiplication but add addition registers with the ports of all components which lengthen the latency of the system.

Figure 5.6: HDL generation for a *Gain* block without chosen *Adaptive Pipelining*

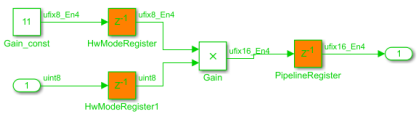

Figure 5.7: HDL generation for a *Gain* block with chosen *Adaptive Pipelining*

Clock-rate Pipelining When *Clock-rate Pipelining* is selected, registers in the designed subsystem are connected to the fastest clock to limit the latency in the system [6]. The delay designed

in the subsystem by using components would not be affected by *Clock-rate Pipelining*. The synthesis tool may plan some resisters to the design during the FPGA programming process when some properties are selected like [6]. These registers have the same data rate with the components on the same path by default. Thus, if there exist multiple paths with different data rates, the registers arranged on the slower path extends the latency from the input of the subsystem to the output. By selecting the*Clock-rate Pipelining*, the registers added in the slower path will be connected with the clock in the faster path to reduce the delay caused by the additional arranged registers. Considering the additional registers added by *Adaptive Pipelining*, choosing *Clock-rate Pipelining* helps to reduce the negative influence of additional delay caused by *Adaptive Pipelining*.

Considering that the matrix multiplication utilizes many multipliers and when N is a large number, the number of physical DSP on the FPGA may not be sufficient. The design should allow multiple multiplications to share one physical DSP. Thus, *Adaptive Pipelining* is selected. There exist multiple data rates in both the classical method and the improved method. To reduce the additional delay caused by *Adaptive Pipelining*, *Clock-rate Pipelining* is also selected.

Chapter 6

Performance Evaluation

6.1 NOILC Simulation

With the knowledge of the provided plant and controller, the impulse response data of the close-loop system from $u_{ff,j}$ to θ_j could be computed directly [71]. With the help of the *impulse* in MATLAB [11], the the impulse response data is presented in Figure 6.1 the convolution matrix **PS** can be constructed based on the Equation 4.7. The three weighting matrices mentioned in Section 4.2 are determined according to the tuning guidelines provided by Barton and Alleyne [25]. After adjustment, $\mathbf{W_e} = I^{N\times N}$, $\mathbf{W_f} = 0.01 \cdot I^{N\times N}$ and $\mathbf{W}_{\delta\mathbf{f}} = 0 \cdot I^{N\times N}$ where $I^{N\times N}$ is a $N \times N$ identity matrix. With the knowledge of three weighting matrices and the convolution matrix, matrices **L** and **Q** becomes known before the system starts working.

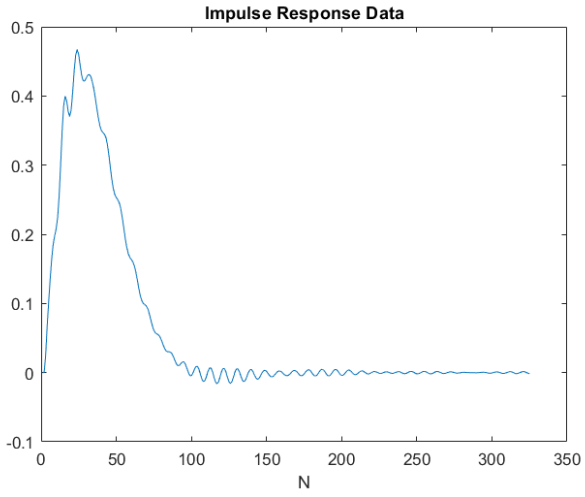

Figure 6.1: The impulse response data of the close-loop system from $u_{ff,j}$ to θ_j

The simulation result of NOILC is shown in Figure 6.2, where the learning process is performed for 20 iterations. The reference position signal is a stair-step signal which does not have to return to the initial position since the input signal e_j reduces to almost zero after convergence. The first figure presents the reference position signal $\theta_{r,j}$ in the blue line and the measured position signal θ_j in the red line which are the input signal and output signal of the feedback control system. The duration of each iteration is 0.020375s and the distance between the destination and the start position is 20um on Z axis. The start position of the first iteration is 0. The start position of the j^{th} $(j > 1)$ iteration is the destination of the $(j-1)^{th}$ iteration. With the help of the feedforward

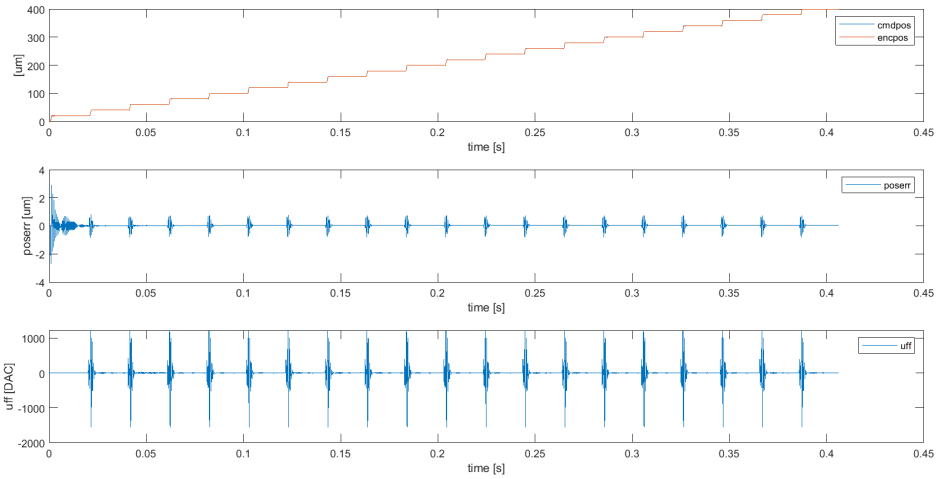

Figure 6.2: The simulation result of NOILC in Simulink

signal output by the NOILC, from the second iteration, the maximum position error, presented in the second figure of Figure 6.2, is decreased to less than 1 um, which is less than the half of the maximum error in the first iteration without the help of the NOILC. When the value of θ_j meets the $\pm 2\%$ of the value difference of $\theta_{r,j}$ and stays in this range, the output is settled and the time is called the settling time of the system [68]. With the help of the NOILC, the settling time reduces from 10.0625 ms in the first iteration to less than 1.8500 ms after the second iteration. The third figure is the feedforward signal $u_{ff,j}$ whose value in the first iteration is zero. The output of ILC is valued and affects the control system from the second iteration. These three figures show that the designed NOILC works and positively affects the performance of the control system. It reduces not only the position error but also assists to speed up the convergence.

The improved model should have the exact same output as the classical model as presented in Figure 6.3 because the two methods are theoretically equivalent based on the equations from Equation 4.29 to Equation 4.31. Two feedforward signals from different models completely overlap with each other showing that the improved model works correctly.

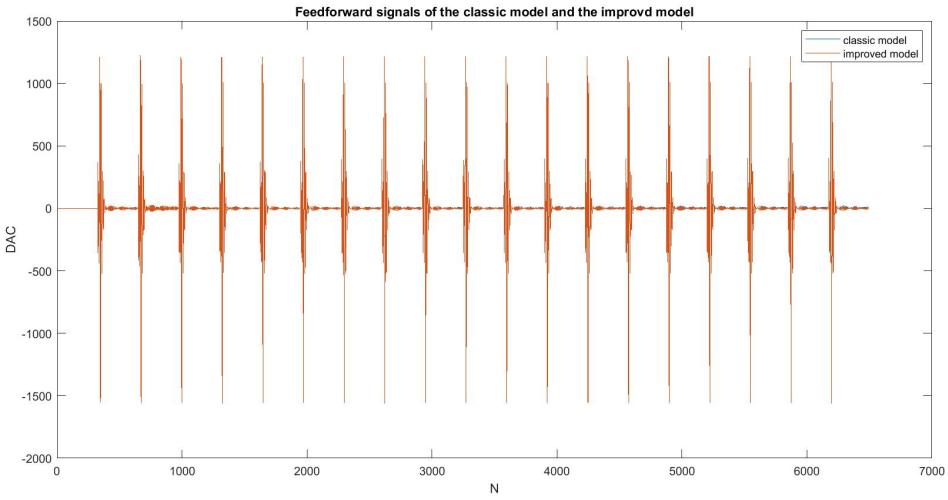

Figure 6.3: Feedforward signals of NOILC model with classic matrix multiplication and improved model

6.2 Resource Utilization Analysis

6.2.1 Classical Matrix Multiplication

Based on the assumption table 4.1 in Section 4.3.3, the required resources for a *Deserializer* and a *Gain* are:

$$\begin{cases} U_{LUT} = bW \cdot N^2 + 1 + 4\lceil log_2(N) \rceil & (6.1) \\ U_{FF} = bW \cdot (2N-1) + 2\lceil log_2(N) \rceil & (6.2) \\ U_{DSP} = N^2. & (6.3) \end{cases}$$

The numbers of required FFs and LUTs has to be integers, the estimated number has to be rounded into the nearest integer which is greater or equal to themselves by $\lceil x \rceil$. With the selected N, the actual resource utilization for the classical model is in the table 6.1. The estimated requirements of LUT, FF and DSP are based on the Equation 6.1, Equation 6.2 and Equation 6.3. From the table, we could see that when a small number of DSPs are required ($N \leq 5$), the actual number of DSPs utilized is the same as the estimated one. With the increase in DSP requirement, the actual DSP utilization becomes slightly less than the computed one. This can happen for more than one reason. Firstly, because the vector and matrix are randomly generated, it cannot avoid generating elements with special values like 0, 1 or powers of 2. The synthesis tool may not arrange DSPs for multiplication with these numbers as LUTs could achieve the functions. The second reason might be that the resource of multipliers in DSPs is not completely utilized. The bit-width for the data is 12 while the size of the multipliers on the chip is 25 $\times$ 18 [1]. When the number of required multipliers rises, the synthesis tool may split part of the 12$\times$12 multiplications into several smaller multiplication groups based on the Baugh-Wooley multiplication algorithm [55] to fully utilise multipliers. By increasing the bit-width to 18, the number of required DSP is increased from 35 to 36 which is the estimated number when N = 6. When N = 14, the number of required DSP increase from 175 to 184. Although, this number is still less than the estimated number (196), the number of required DSP rises when the bit-width becomes larger. The third reason might be that the multiplication is transferred into addition by Karatsuba-Of-man algorithm [51] [60]. With the help of this algorithm, the karatsuba algorithm reduces the multiplication size by half with each use, reducing the cost of DSPs.

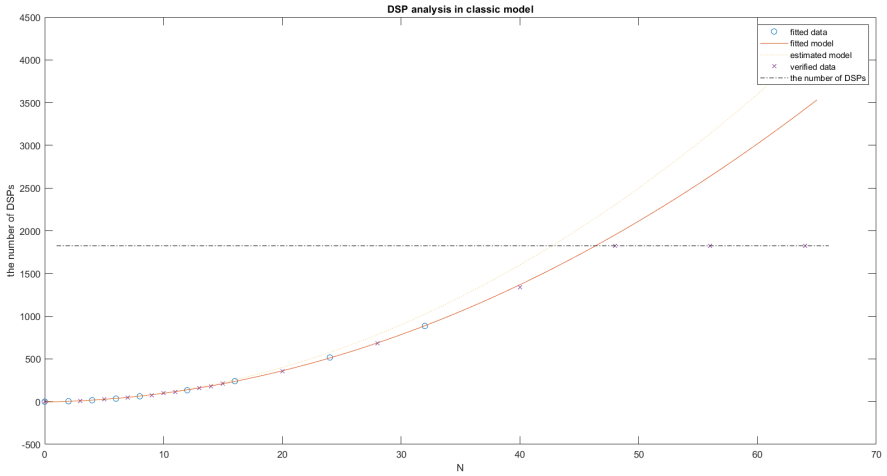

Figure 6.4: The DSP utilization of classical matrix multiplication

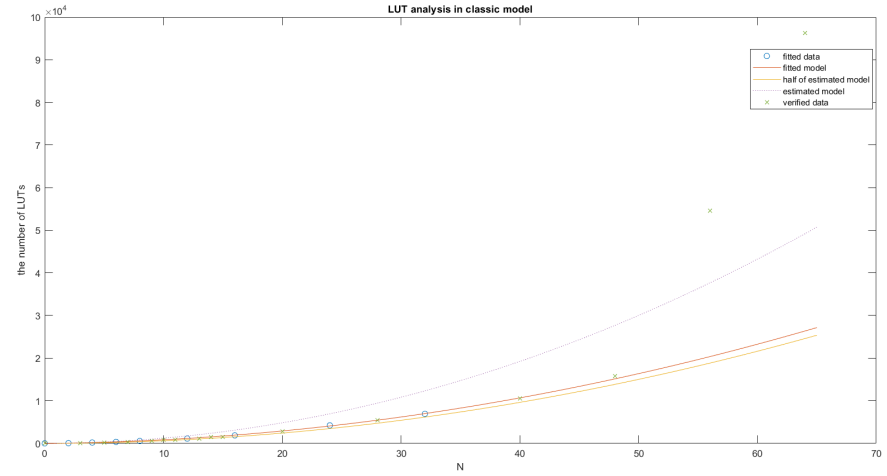

Figure 6.5: The LUT utilization of classical matrix multiplication

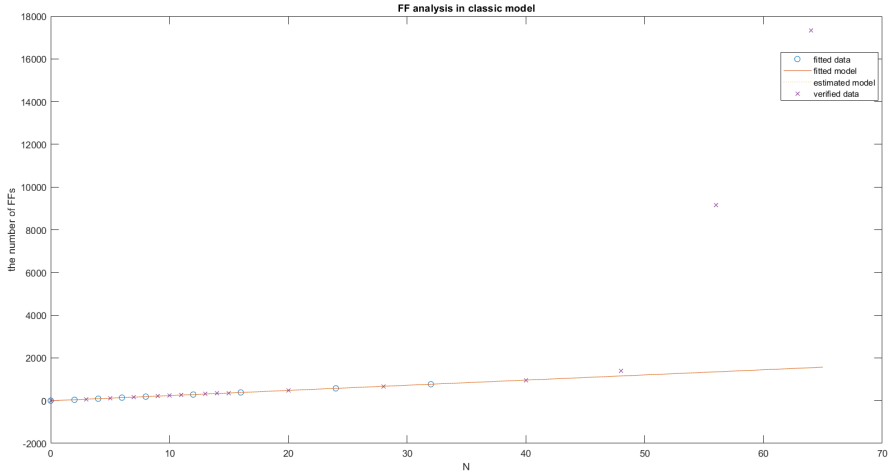

Figure 6.6: The FF utilization of classical matrix multiplication

Table 6.1: Resource utilization of classical matrix multiplication

N		2	3	4	5	6	7	8	9	10	11	12	13
LUT	Estimated	53	117	201	313	445	601	781	989	1217	1469	1745	2045
	Actual	53	77	180	187	343	343	552	550	811	804	1116	1105
FF	Estimated	38	64	88	114	138	162	186	212	236	260	284	308
	Actual	39	65	89	115	139	163	187	213	237	261	285	309
DSP	Estimated	4	9	16	25	36	49	64	81	100	121	144	169
	Actual	4	9	16	25	35	48	62	76	99	111	134	158

N		14	15	16	20	24	28	32	40	48	56	64	
LUT	Estimated	2369	2717	3089	4821	6933	9429	12309	19225	27673	37657	49177	
	Actual	1470	1455	1872	2843	4210	5354	6924	10579	15766	54570	96277	
FF	Estimated	332	356	380	478	574	670	766	960	1152	1344	1536	
	Actual	333	357	381	479	575	671	768	961	1385	9157	17323	
DSP	Estimated	196	225	256	400	576	784	1024	1600	2304	3136	4096	
	Actual	175	210	240	352	517	680	885	1338	1824	1824	1824	

Figure 6.4 presenting the relationship of N with the number of utilized DSPs. The estimated model is based on the Equation 6.3. Even though, the number of actual utilized DSPs is less than the estimated number. The possible causes of this phenomenon were discussed in the preceding paragraph. When the demand for DSPs exceeds the available DSP number on the chips, the number of actually used DSPs does not vary. The number of available DSPs here is 1824 represented in the dash-dotted line. When N is larger than 42, the actual utilized DSP would not be changed since it meets the maximum. Thus, all data with N equals a power of 2 ($N < 42$) are selected as a fitted set and the others are selected for verification.

Quadratic model in polynomial regression models is selected while doing least squares. The reason why to choose the quadratic model for fitting is that the number of actual utilized DSPs is a little less than the estimated model (N^2) but the increasing trend does not vary a lot. A data set to be fitted contains n points (x_i, y_i), $n = 1, 2, \ldots, n$, where x_i is the independent variable and y_i is the dependent variable. Residuals (r_i) are the difference between observation values (y_i) and the fitted values $f(x_i, \boldsymbol{\beta})$ as Equation 6.4 presented [36]. Least squares finds the matched regression function for data set by minimising the sum of the squares of the residuals (S) [36].

$$r_i = y_i - f(x_i, \boldsymbol{\beta}) \tag{6.4}$$

$$S = \sum_{i}^{n} r_i^2 \tag{6.5}$$

$$RMSD = \sqrt{\frac{\sum_k (fit[k] - verification[k])^2}{length}} \tag{6.6}$$

$$NRMSD = \frac{RMSD}{verification} \tag{6.7}$$

MATLAB provides a function *fit* [10] helping to do the polynomial regression. Thus, a lot calculation processes of least squares could be saved. The fitted model found by MATLAB ($U_{DSP} = 0.8027 \cdot N^2 + 2.206 \cdot N - 4.253$) is the red line in the Figure 6.4 which is almost fitted with the experimental data in table 6.1 for all N less than 42. This model computes the the cost of DSP with the increase of N when the bit-width is 12. To calculate the errors of the fitted model, in the verified set only the parameters when $N < 42$ are considered. The root-mean-square deviation (RMSD) between the fitted model and the verified data is 8.7498 calculated based on the Equation where k is the value of N in the verified set and the *length* is the total number of verified sets. The Normalization root-mean-square deviation (NRMSD) is 0.0160 calculated based

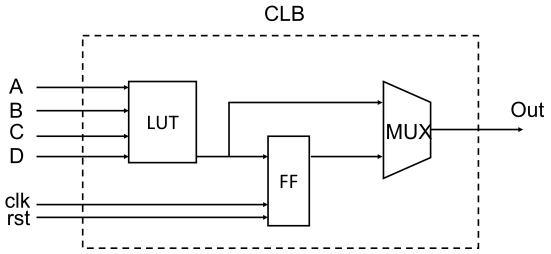

Figure 6.7: The structure of a configurable logical block on FPGA

on the Equation 6.7 where $verification$ is the mean of verified data. The error of the fitted model is 1.6% which is acceptable. Overall, for the classical matrix multiplication, the actual utilized DSPs quadratically increase with the scalar N and the fitted model is dependable.

The estimated cost of LUT for the classical implementation is based on the Equation 6.1. However, compared with the data in table 6.1, the actual utilized LUT is roughly the half of the expectation for $N \leq 42$. The reason of this difference might be the configurable logic block (CLB) in FPGA which is constructed by a LUT, a FF and a MUX. The structure of CLB is shown in Figure 6.2.1 where the LUT implements the combinational logical function, FF stores the output of LUT and MUX implements the selection logic. The CLB allows one single LUT to implement two logic functions. Without the construction of CLB, the LUT in the Figure with 4 inputs could only achieve the logic with at most 4 inputs to 8 inputs. For an eight inputs logic, the inputs are split into two groups, four inputs as a group. The logical output of one group is stored in the FF and then output to the MUX and the logical output of the other group is output to the MUX directly for the selection. Due to the structure of CLB, the number of inputs to the LUT increases from at most 4 to at most 8 in this example. The number of actual utilized LUT is decreased.

By selecting the same fitted set as the DSP analysis, the rest sets are selected for verification. When there are enough DSPs on the chip, the fitted model computed by using the *fit* function ($U_{LUT} = 6.04 \cdot N^2 + 25.67 \cdot N - 24.93$) is fitted with the verification data as presented in Figure 6.5. The RMSD for the fitted model of required LUT is 115.3192 and the NRMSD is 0.0096 for $N < 42$. However, when the number of DSPs is insufficient ($N > 42$), the actual number of LUTs used skyrockets because additional multipliers are constructed with CLBs by the synthesis tool. Thus, a lot LUTs are consumed. As a result, the usage of LUTs grows to a very high level. If the number N continues to grow, the LUTs on the chip will all be used. With the increase of N, when the number of DSP is sufficient, the growth of the actual utilized LUT should be increased by a power of 2 since the fitted model is almost same with the half of the estimated model when $N < 42$. In addition, since the NRMSD of the fitted model is small enough, the output of the fitted model is almost same with the number of actual utilized LUT in the classical design.

The number of required FFs from computation is based on the Equation 6.2. The actual number of required FFs is almost the same as the estimated number when the number of DSP is sufficient. After doing the polynomial regression with the fitted data in Figure 6.6, the fitted model is almost the same as the estimated model and fitted with all verification data when $N < 42$. The RMSD of the fitted model is 1.5849 and the NRMSD is 0.000792. The differences between the fitted model and the verified data are small enough to be neglected. Same with the situation of LUTs, when the number of DSP is insufficient, the number of actual required FFs rises dramatically. The required FF linearly grows with the increase of N when DSP is sufficient. When the number of DSPs is insufficient, the number of required FFs and LUTs skyrockets, soon consuming all available chip resources.

Table 6.2: Resource utilization of improved matrix multiplication model

	N	2	3	4	5	6	7	8	9	10	11	12	13	14
LUT	Estimated	80	122	158	200	236	272	308	350	386	422	458	494	530
	Actual	66	64	106	103	157	139	221	194	273	231	321	267	369
FF	Estimated	63	102	138	177	213	249	285	324	360	396	432	468	504
	Actual	65	110	146	196	244	292	324	359	399	435	473	510	551
DSP	Estimated	1	1	1	1	1	1	1	1	1	1	1	1	1
	Actual	1	1	1	1	1	1	1	1	1	1	1	1	1
	N	15	16	20	24	28	32	40	48	56	64	80	96	112
LUT	Estimated	566	602	752	896	1040	1184	1478	1766	2054	2342	2924	3500	4076
	Actual	303	415	519	615	710	805	778	924	1069	1210	1517	1804	2107
FF	Estimated	540	576	723	867	1011	1155	1446	1734	2022	2310	2889	3465	4041
	Actual	586	626	769	917	1065	1214	1513	1809	2105	2401	2997	3604	4199
DSP	Estimated	1	1	1	1	1	1	1	1	1	1	1	1	1
	Actual	1	1	1	1	1	1	1	1	1	1	1	1	1
	N	128	160	192	224	256	320	384	448	512	640	768	896	1024
LUT	Estimated	4652	5810	6962	8114	9266	11576	13880	16184	18488	23102	27710	32318	36926
	Actual	3164	3951	3581	4175	6300	7883	7149	10984	12543	11834	18820	21940	25062
FF	Estimated	4617	5772	6924	8076	9228	11535	13839	16143	18447	23058	27666	32274	36882
	Actual	4779	5964	7147	8333	9521	11895	14254	16634	19010	23739	28495	33249	38003
DSP	Estimated	1	1	1	1	1	1	1	1	1	1	1	1	1
	Actual	1	1	1	1	1	1	1	1	1	1	1	1	1

6.2.2 Improved Matrix Multiplication

The analysis of the improved model contains a multiplier, a *Deserializer*, an adder, a *UnitDelay*, two switches, a counter and a constant 0 which already mentioned in Section 5.4. All the tested data are presented in table 6.2. All data sets with N equals a power of 2 are selected as a fitted set and the others are selected for verification. Based on the table 4.2 in Section 4.4.3, the calculated required resources for the improved model are:

$$\begin{cases} U_{LUT} = 3bW \cdot N + 6\lceil log_2(N) \rceil + 2 & (6.8) \\ U_{FF} = bW(2N-1) + 3\lceil log_2(N) \rceil & (6.9) \\ U_{DSP} = 1. & (6.10) \end{cases}$$

The estimated number of DSP is only 1 which should not increase with the growth of N. Based on the table 6.2, the number of required DSP remains 1 for all N. The fitted model is also the same as the estimated model fitting all the verification data for the number of DSP utilization in the improved model.

The estimated model of the cost of LUTs in the improved model is based on the Equation 6.8. The actual cost of LUT is around two thirds of the estimated number which might be caused by the CLB blocks mentioned above. The reason for this mismatch is that the two switches and adder in the improved model employ the most LUTs in the design. The CLB structure, on the other hand, would not reduce the number of required LUTs required by the $Switch2$ because the number of its inputs is the same as the number of its outputs, and these output signals are exported simultaneously. The required LUTs of $Switch2$ cannot be reduced. The fitted model in Figure 6.9 overlaps with two thirds of the estimated model but does not perfectly fit with the verification data. There exist some points underneath the fitted model but most data fits with the fitted model. The fitted model ($U_{LUT} = 24.24 \cdot N - 171.4$) is a linear model, however, according to the computation, there also exist logarithmic calculations for LUT utilization. The fitted model is overlapped with the two third of the estimated model. The linear model cannot perfectly fit with the validation data. In addition, there exist four points which are far away from the fitted model. The data does not alter after multiple repetitions of the same test. Thus, the gaps between these points and the fitted model is not caused by the values of randomly generated vectors and matrices. Although I am not sure about why these problematic points are generated, the fitted model can still be used to predict the cost of LUTs. The reason is that these points are

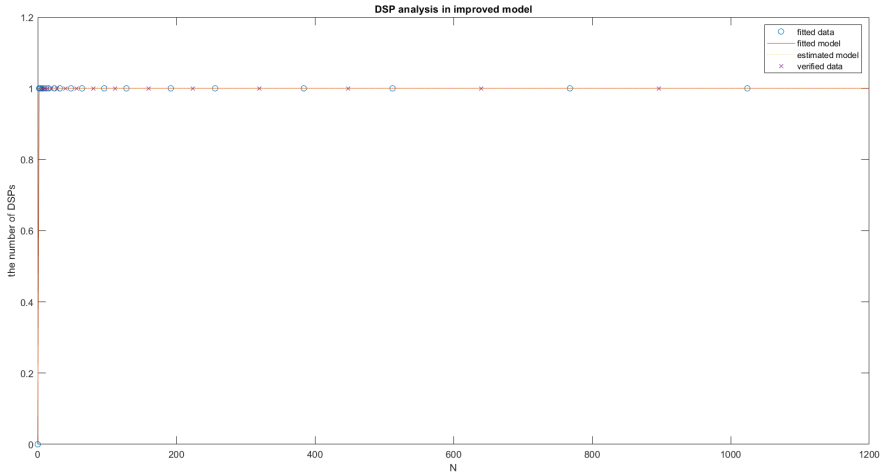

Figure 6.8: The DSP utilization of improved matrix multiplication

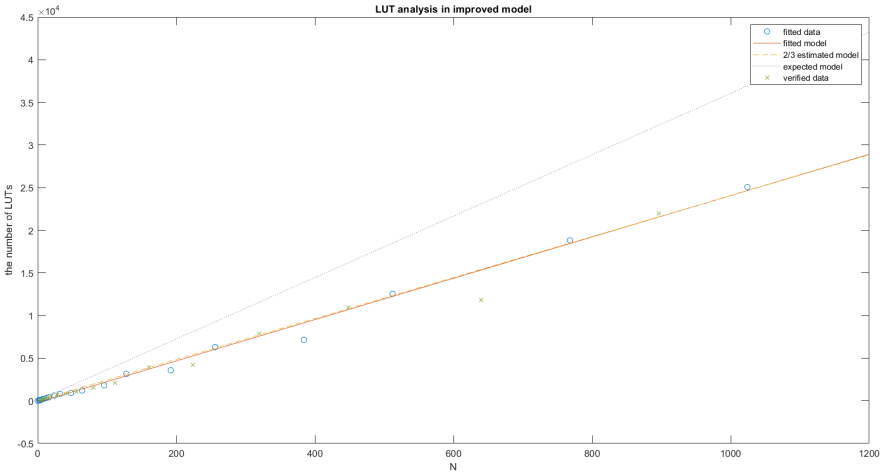

Figure 6.9: The LUT utilization of improved matrix multiplication

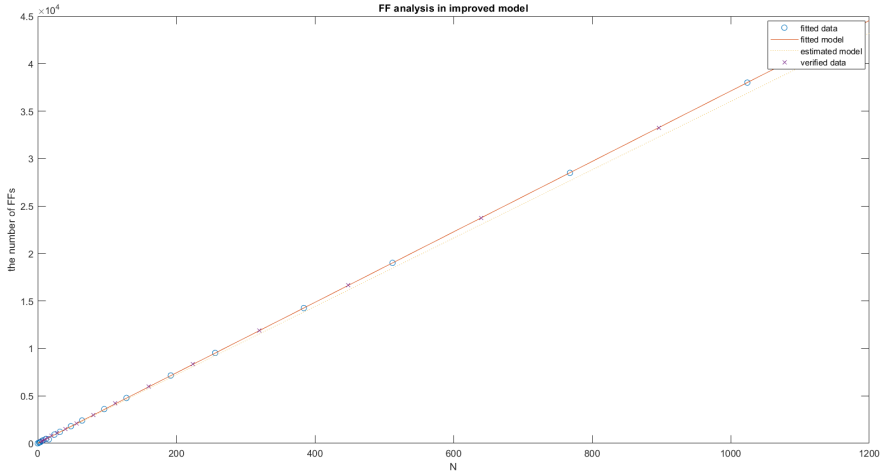

Figure 6.10: The FF utilization of improved matrix multiplication

less than the estimated value. The implementation based on this prediction would not influence the performance of a project. Due to these problematic points, the error of this model is larger than other models. The RMSD of the usage of LUT in the improved design is 947.0165 and the NRMSD is 0.2217. Because the NRMSD is larger than 20%, this fitted model cannot accurately predict the actual LUT usage. Even though there exist fluctuations, in the improved model, the number of utilized LUT linearly increases with the growth of N.

The estimated model of FF in the improved model is based on the Equation 6.9. The data in table 6.1 shows that the actual utilized FF is a little higher than the estimated number and the gap rises with the increase of scalar N. The reason might be that *AdaptivePipelining* adds some additional registers at the input and the output ports of the components which are represented by FFs. In the classical design, there are only two components. The influence of the additional registers is not apparent. But in the improved design which has 8 components, the influence of additional registers becomes apparent. After fitting a model based on the selected data, the models and data are presented in Figure 6.10 where the fitted model almost fits with the verification data. The RMSD of the fitted model is 20.5661 and the NRMSD is 0.0029, which are so minor that can be ignored. No matter for the fitted model or the computed model, the number of utilized FF is linearly increasing with the scalar N. But the gap between the fitted model and the estimated model becomes larger with the growth of N due to the FFs connected between components.

6.2.3 The Comparison of the classical Model with the Improved Model

When the number of DSPs on the chip is sufficient, the usages of LUT and DSP in the improved design are significantly lower than those in the classical design. The number of utilized FFs in the improved design is slightly more than the number of utilized FFs in the classical design when DSP is sufficient. The reason might be the influence of *Adaptive Pipelining* which adds additional registers in the design. When $N = 40$, the number of used LUTs in the classical design is similar to the number of used LUTs in the improved design when $N = 448$. The gaps in the use of LUT and DSP in the two models widen as N grows. This is because the growth pattern of the utilized DSP is reduced from quadratic growth to constant, and the growth pattern of the used LUT is lowered to linear. There will be no spikes in LUT and FF usage due to the DSP's resources running out because the DSP utilization is controlled to a range which can be decided by designers.

Although the classical design utilized significantly more resources than the improved model, the NOILC implementation would double the utilization for the same N in the classical design since NOILC contains two matrix multiplications. The NOILC implementation with the improved model, on the other hand, requires roughly the same resources as the single matrix multiplication test, plus a multiplier and an adder. As a result, the improved model has much higher resource utilization efficiency, allowing a NOILC system with a larger N to be executed on an FPGA platform.

6.3 NOILC implementation on FPGA

To test the performance of NOILC when $N = 325$ on the hardware, MATLAB provides an interface, FPGA-in-the-Loop (FIL), which connecting the simulator with the real hardware to verify the function of HDL code. MATLAB generates a simulation model on the register transfer level (RTL) based on the design in Simulink during the HDL generating process. During the process of HDL generation, MATLAB creates a simulation model on the RTL based on the HDL code.

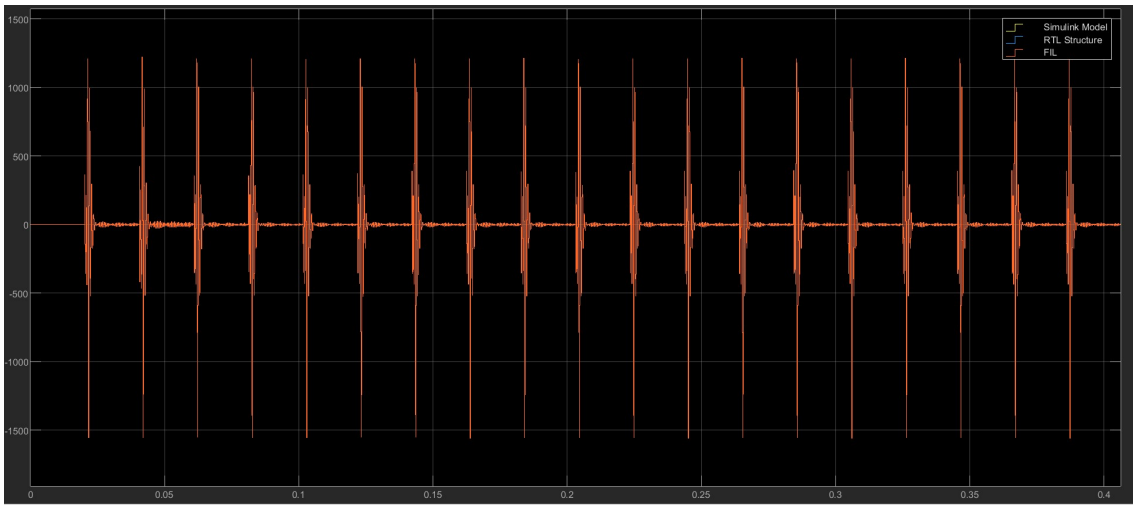

Figure 6.11: The testing result of the improved NOILC in the hardware by FPGA-in-the-loop

$$\begin{cases} U_{LUT} = 2 \times \dfrac{bW \cdot N^2 + 1 + 4\lceil log_2(N) \rceil}{2} + bW = 2542837 & (6.11) \\ U_{FF} = 2 \times (bW \cdot (2N-1) + 2\lceil log_2(N) \rceil) = 31188 & (6.12) \\ U_{DSP} = 2N^2 = 211250 & (6.13) \end{cases}$$

By providing the same inputs of the simulation model in Simulink, the outputs signals from the simulation, the model on the RTL and the verification in FIL are presented in Figure 6.11. Three signals are overlapped with each other showing that these three signals are exactly the same. The function of the improved model implemented on FPGA is correct. Based on the tests and model fittings of matrix multiplication models, the resources of the NOILC implemented by improved method and classical method is able to be predicted. The bit-widths of the signals in the NOILC system are 24 bits as presented in Section 5.2.3. When the NOILC is implemented with the classic method, the cost the double of the vector-matrix multiplication plus an additional adder. According to the fitted model of LUTs cost implemented the classic method, the predicted LUTs cost is half of the estimated model described by Equation 6.1. The fitted model of FFs is almost same with the estimated model. Thus, Equation 6.2 could predict the FF cost of vector-matrix multiplication. The fitted model of DSP cost is less than the estimated model. When Equation 6.3 is used to predict, the actual DSP cost will not exceed the prediction. The predicted cost is presented in the Equation 6.11, Equation 6.12 and Equation 6.13. The cost of DSP should be 211250, the cost of LUT should be 2535061 and the cost of FF should be 31188.

$$\begin{cases} U_{LUT} = \lceil \dfrac{2}{3}(3bW \cdot N + 6\lceil log_2(N) \rceil + 2) \rceil + bW = 15662 & (6.14) \\ U_{FF} = bW(2N-1) + 3\lceil log_2(N) \rceil = 15603 & (6.15) \\ U_{DSP} = 2. & (6.16) \end{cases}$$

The cost of NOILC implemented by the improved method is same the cost of vector-matrix multiplication but pulse an additional adder and an additional multiplier. The fitted model of LUT cost for the improved method is around two third of the estimated model. The predicted LUTs cost is two third of the estimated model described by Equation 6.8 plus an adder (bW). The cost of FF can be predicted by the estimated model (Equation 6.9) and there should be 2 multipliers in NOILC system. The predicted cost is presented in the Equation 6.14, Equation 6.15 and Equation 6.16. The predicted cost of DSP decreases from 211250 to 2, from an impossible

value to an achievable value. The predicted cost of LUT reduces from 2535061 to 15662 which is only 0.62% of the classical method. The predicted cost of FF reduces from 31188 to 15662 around half of the classical method. The costs of LUT, FF and DSP all decreases a lot with the improved method.

Chapter 7

Conclusion

The project focused on implementing a norm optimal iterative learning control algorithm on FPGA through FPGA programming with MATLAB. The NOILC system is created in MATLAB Simulink using the available feedback control model in ASM to test its performance. This research proposes an improved NOILC model that generates the same feedforward signal as the classical NOILC model but with significantly higher resource utilization efficiency. The growth trend of the required number of DSPs is reduced from the quadratic growth to a constant. The growth trend of the required number of LUTs is decreased from the quadratic growth to the linear growth. The utilization of FFs in the improved design is similar to the utilization of FFs in the classical model. Except for the usage of LUT in the improved design which constrains some fluctuation, the NRMSD of the other trained models, either in the classical designs or improved designs, are less than 2%. These trained models can be utilized to predict the resource utilization for matrix multiplications. With these prediction model, the resource utilization of NOILC with these two implementation methods could be realized in the simulation step rather than known until the HDL project is generated by MATLAB. The time required to generate HDL projects varies for different projects. Hours may be spent for a large project. In addition, designers are able to realize that whether the resource on the hardware is enough to process the designed systems and alter the designs in time. Thus, the prediction models would save a lot time for designers. The improved model has been validated in simulation as well as on hardware. Because fewer multipliers are utilized, the high-complexity algorithm NOILC can be implemented on an FPGA with limited DSP resources, even though the number of elements in one iteration is a large number.

There are some unavoidable issues with FPGA programming with MATLAB, ranging from property settings to performance analysis. Inappropriate property settings may cause the model to perform poorly, such as longer latency and more resource utilization, when operating on hardware. The clock rate specified via MATLAB is limited to a specific range (between 5MHz and 500MHz) rather than the full range that FPGAs can provide. Due to the multiplication operated serially, for a project with a very large number of elements in one iteration like 10 thousand samples, the clock rate of the current Zedboard cannot support the execution as fast as the project needs. FPGAs with faster clock rates and more resources are available, although they are often much more expensive. Although the current model does not yet operate a real motor via an FPGA, it does allow for the implementation of a NOILC using a resource-constrained FPGA.

The FPGA programming provided by MATLAB allows engineers who do not familiar with hardware programming to create hardware projects and load them on hardware platforms. In addition, engineers are able to monitor the feedback signals from the hardware by the interface provided by MATLAB. This approach decreases the difficulty for designers who are unfamiliar with hardware languages to implement models.

Bibliography

[1] 7 series dsp48e1 slice user guide. https://docs.xilinx.com/v/u/en-US/ug479_7Series_DSP48E1. 33

[2] Adaptive pipelining in matlab. https://nl.mathworks.com/help/hdlcoder/ug/adaptive-pipelining_bve57q1-1.html. 28

[3] Add, subtract, sum of elements, sum in simulink. https://nl.mathworks.com/help/simulink/slref/add.html. 26

[4] Board products in xilinx. https://www.xilinx.com/products/boards-and-kits.html. 13

[5] Buffer in simulink. https://nl.mathworks.com/help/dsp/ref/buffer.html. 24

[6] Clock-rate pipelining in matlab. https://nl.mathworks.com/help/hdlcoder/ug/clock-rate-pipelining.html. 28, 29

[7] Constant in simulink. https://nl.mathworks.com/help/simulink/slref/constant.html. 26

[8] Counter in simulink. https://nl.mathworks.com/help/dsp/ref/counter.html. 26

[9] Deserializer1d in simulink. https://nl.mathworks.com/help/hdlcoder/ref/deserializer1d.html. 24

[10] Function fit in matlab. https://nl.mathworks.com/help/curvefit/fit.html#bto2vuv-1_vh. 35

[11] Function impulse in matlab. https://nl.mathworks.com/help/control/ref/lti.impulse.html. 31

[12] Gain in simulink. https://nl.mathworks.com/help/simulink/slref/gain.html. 24

[13] Product, matrix multiply in simulink. https://nl.mathworks.com/help/simulink/slref/product.html. 26

[14] Serializer1d in simulink. https://nl.mathworks.com/help/hdlcoder/ref/serializer1d.html. 24, 27

[15] Switch in simulink. https://nl.mathworks.com/help/simulink/slref/switch.html. 26

[16] Unit delay in simulink. https://nl.mathworks.com/help/simulink/slref/unitdelay.html. 26

[17] Unleash the unparalleled power and flexibility of zynq ultrascale+ mpsocs. https://www.xilinx.com/content/dam/xilinx/support/documents/white_papers/wp470-ultrascale-plus-power-flexibility.pdf. 1

[18] Zedboard in xilinx. https://www.xilinx.com/products/boards-and-kits/1-8dyf-11.html. 23

[19] Zynq-7000 soc family. https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html#productTable. 13, 23

[20] Peter Alfke, Ivo Bolsens, Bill Carter, Mike Santarini, and Steve Trimberger. It's an fpga! *IEEE Solid-State Circuits Magazine*, 3(4):15–20, 2011. 1

[21] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):367–432, sep 1995. 6

[22] Abbes Amira, Ahmed Bouridane, and Peter Milligan. Accelerating matrix product on reconfigurable hardware for signal processing. In *International Conference on Field Programmable Logic and Applications*, pages 101–111. Springer, 2001. 6

[23] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *2009 international conference on field programmable logic and applications*, pages 126–131. IEEE, 2009. 1

[24] Ahsan Javed Awan. *FPGA Based Implementation of Norm Optimal Iterative Learning Control*. PhD thesis, 06 2012. 6

[25] Kira Barton and A.G. Alleyne. Norm optimal ilc with time-varying weighting matrices. pages 264 – 270, 07 2009. 31

[26] Kira Barton, Jeroen Van De Wijdeven, Andrew Alleyne, Okko Bosgra, and Maarten Steinbuch. Norm optimal cross-coupled iterative learning control. In *2008 47th IEEE Conference on Decision and Control*, pages 3020–3025. IEEE, 2008. 11

[27] Kira L. Barton and Andrew G. Alleyne. A norm optimal approach to time-varying ilc with application to a multi-axis robotic testbed. *IEEE Transactions on Control Systems Technology*, 19(1):166–180, 2011. 2

[28] Frede Blaabjerg. *Control of Power Electronic Converters and Systems: Volume 2*, volume 2. Academic Press, 2018. 7

[29] JJ Bolder. Flexibility and robustness in iterative learning control: with applications to industrial printers. 2015. 10, 11

[30] Douglas A Bristow, Marina Tharayil, and Andrew G Alleyne. A survey of iterative learning control. *IEEE control systems magazine*, 26(3):96–114, 2006. 5

[31] Scott J Campbell and Sunil P Khatri. Resource and delay efficient matrix multiplication using newer fpga devices. In *Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pages 308–311, 2006. 6

[32] Lynn Elliot Cannon. *A cellular computer to implement the Kalman filter algorithm*. Montana State University, 1969. 7

[33] Jaeyoung Choi, David W Walker, and Jack J Dongarra. Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994. 6

[34] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. Understanding performance differences of fpgas and gpus. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96, 2018. 1

[35] Charles R Cutler and Brian L Ramaker. Dynamic matrix control?? a computer control algorithm. In *joint automatic control conference*, number 17, page 72, 1980. 2

[36] Frederik Michel Dekking, Cornelis Kraaikamp, Hendrik Paul Lopuhaä, and Ludolf Erwin Meester. *A Modern Introduction to Probability and Statistics: Understanding why and how*, volume 488. Springer, 2005. 35

[37] Yong Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point fpga matrix multiplication. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, FPGA '05, page 86–95, New York, NY, USA, 2005. Association for Computing Machinery. 7

[38] Weinmann Dr. Florian Dietz. Weinmann develops life-saving transport ventilator using model-based design, 2015. 8

[39] Qiang Fei, Yongting Deng, Hongwen Li, Jing Liu, and Meng Shao. Speed ripple minimization of permanent magnet synchronous motor based on model predictive and iterative learning controls. *IEEE Access*, 7:31791–31800, 2019. 5

[40] G.C Fox, S.W Otto, and A.J.G Hey. Matrix algorithms on a hypercube i: Matrix multiplication. *Parallel Computing*, 4(1):17–31, 1987. 6

[41] Katsuhisa Furuta and Manop Wongsaisuwan. Closed-form solutions to discrete-time lq optimal control and disturbance attenuation. *Systems Control Letters*, 20(6):427–437, 1993. 2

[42] Murray Garden. Learning control of actuators in control systems, January 12 1971. US Patent 3,555,252. 2

[43] Graham Clifford Goodwin, Stefan F Graebe, Mario E Salgado, et al. *Control system design*, volume 240. Prentice Hall Upper Saddle River, 2001. 1

[44] William R Graham. The parallel, pipeline, and conventional computer. Technical report, RAND CORP SANTA MONICA CALIF, 1969. 6

[45] Svante Gunnarsson and Mikael Norrlöf. On the design of ilc algorithms using optimization. *Automatica*, 37(12):2011–2016, 2001. 2

[46] M. Haldar, A. Nayak, N. Shenoy, A. Choudhary, and P. Banerjee. Fpga hardware synthesis from matlab. In *VLSI Design 2001. Fourteenth International Conference on VLSI Design*, pages 299–304, 2001. 7

[47] K Hwang and A Faye. Computer architecture and parallel processing. 1 1984. 6

[48] Ju-Wook Jang, S.B. Choi, and V.K. Prasanna. Energy- and time-efficient matrix multiplication on fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(11):1305–1319, 2005. 6

[49] Zhihao Jiang and Rahul Mangharam. University of pennsylvania develops electrophysiological heart model for real-time closed-loop testing of pacemakers, 2013. 7

[50] Xu Jin. Fault-tolerant iterative learning control for mobile robots non-repetitive trajectory tracking with output constraints. *Automatica*, 94:63–71, 2018. 2

[51] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962. 33

[52] Zaher M. Kassas. Methodologies for implementing fpga-based control systems. *IFAC Proceedings Volumes*, 44(1):9911–9916, 2011. 18th IFAC World Congress. 1

[53] Srinidhi Kestur, John D. Davis, and Eric S. Chung. Towards a universal fpga matrix-vector multiplication architecture. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 9–16, 2012. 7

[54] Joo-Young Kim. Chapter five - fpga based neural network accelerators. In Shiho Kim and Ganesh Chandra Deka, editors, *Hardware Accelerator Systems for Artificial Intelligence and Machine Learning*, volume 122 of *Advances in Computers*, pages 135–165. Elsevier, 2021. 7

[55] Ananda Kiran and Navdeep Prashar. Fpga implementation of high speed baugh-wooley multiplier using decomposition logic. *Emerging Trends in Electrical, Electronics & Instrumentation Engineering: An international Journal,(EEIEJ)*, 2(3), 2015. 33

[56] HT Kung and Charles E Leiserson. Systolic arrays (for vlsi). In *Sparse Matrix Proceedings 1978*, volume 1, pages 256–282. Society for industrial and applied mathematics, 1979. 6

[57] Colin Yu Lin, Hayden Kwok-Hay So, and Philip H.W. Leong. A model for matrix multiplication performance on fpgas. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 305–310, 2011. 7

[58] Nicolas Minorsky. Directional stability of automatically steered bodies. *Journal of the American Society for Naval Engineers*, 34(2):280–309, 1922. 2

[59] Saeid Mokhatab and William A. Poe. Chapter 14 - process control fundamentals. In Saeid Mokhatab and William A. Poe, editors, *Handbook of Natural Gas Transmission and Processing (Second Edition)*, pages 473–509. Gulf Professional Publishing, Boston, second edition edition, 2012. 2

[60] Peter L Montgomery. Five, six, and seven-term karatsuba-like formulae. *IEEE Transactions on Computers*, 54(3):362–369, 2005. 33

[61] Kevin L Moore. Iterative learning control for deterministic systems. 2012. 2

[62] Robert Oshana. *DSP for Embedded and Real-time Systems*. Elsevier, 2012. 7

[63] David H. Owens, Christopher T. Freeman, and Thanh Van Dinh. Norm-optimal iterative learning control with intermediate point weighting: Theory, algorithms, and experimental evaluation. *IEEE Transactions on Control Systems Technology*, 21(3):999–1007, 2013. 2

[64] Feng Qiu, Shinichiro Michizono, Toshihiro Matsumoto, and Takako Miura. Combined disturbance-observer-based control and iterative learning control design for pulsed superconducting radio frequency cavities. *Nuclear Science and Techniques*, 32(6):1–12, 2021. 5

[65] S Srinivasulu Raju, TS Darshan, and B Nagendra. Design of quadratic dynamic matrix control for driven pendulum system. *International Journal of Electronics and Communication Engineering*, 5(3):363–370, 2012. 2

[66] Eric Roesler and Brent Nelson. Novel optimizations for hardware floating-point units in a modern fpga architecture. In *International Conference on Field Programmable Logic and Applications*, pages 637–646. Springer, 2002. 7

[67] Yam P. Siwakoti and Graham E. Town. Design of fpga-controlled power electronics and drives using matlab simulink. In *2013 IEEE ECCE Asia Downunder*, pages 571–577, 2013. 7

[68] Teng-Tiow Tay, Iven Mareels, and John B Moore. *High performance control*. Springer Science & Business Media, 1998. 32

[69] Keith D Underwood and K Scott Hemmert. Closing the gap: Cpu and fpga trends in sustainable floating-point blas performance. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 219–228. IEEE, 2004. 7

[70] Robert A Van De Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997. 6, 7

[71] Gijs van der Veen, Joep Stokkermans, Noud Mooren, and Tom Oomen. How learning control supports industry 4.0 in semiconductor manufacturing. In *ASPE Spring Topical Meeting 2020: Design and Control of Precision Mechatronic Systems-Virtual Meeting*, pages 1–5. American Society of Precision Engineering (ASPE), 2020. 31

[72] H.J.B. van Deursen. Multivariable Iterative Learning Control with basis functions: performance enhancement of an industrial flatbed printer. Master's thesis, Eindhoven University of Technology, 2015. 2, 10

[73] Robin van Es & Dragan Kostić. Lecture slits of norm optimal lifted iterative learning control in asm, January 2010. 11, 12

[74] Jurgen van Zundert, Joost Bolder, Sjirk Koekebakker, and Tom Oomen. Resource efficient ilc: Enabling large tasks on an industrial position-dependent flatbed printer. *IFAC-PapersOnLine*, 49(21):567–574, 2016. 7th IFAC Symposium on Mechatronic Systems MECHATRONICS 2016. 5

[75] Jurgen van Zundert, Joost Bolder, Sjirk Koekebakker, and Tom Oomen. Resource-efficient ilc for lti/ltv systems through lq tracking and stable inversion: Enabling large feedforward tasks on a position-dependent printer. *Mechatronics*, 38:76–90, 2016. 2, 5

[76] Raphael Yuster and Uri Zwick. Fast sparse matrix multiplication. *ACM Trans. Algorithms*, 1(1):2–13, jul 2005. 13