A SCALABLE AND COMPACT LINEAR SOLVER WITH A FOCUS ON MODEL PREDICTIVE CONTROL

Ong Shen Hoong, Kevin

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirements for the degree of
Masters of Engineering

2014

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research and has not been submitted for a higher degree to any other University or Institution.

....	....
Date	Ong Shen Hoong, Kevin

Acknowledgments

I would like to express my sincere gratitude and appreciation to my supervisors, Prof Suhaib A.Fahmy(SCE) and Prof Ling Keck-Voon(EEE) for the invaluable experience as your research student. To Prof Suhaib, for your patience, constructive and forward thinking comments especially when my research direction tilted towards a different outcome. To Prof Ling for teaching me how to crystallize my thoughts, proper technical writing and keeping me on my toes at all times. In addition, a special mention to Prof Ling for the rewarding experience gained from multiple roles during my secondment for the A*STAR funded Embedded & Hybrid Systems II Programme (2008-2010).

Within NTU, I would like to thank various people at Centre for High Performance Embedded Systems (CHiPES) and EEE Control Engineering Lab. From CHiPES, Jeremiah Chua for his prompt IT system support and Vipin Kizheppatt for his helpful advice in digital hardware designs for FPGA. Not forgetting the MSc students(Nithin, Zain and Rakesh) who were willing to be my listening ear during the off-peak hours in CHiPES. From the Control lab, Thuy Dang Van and Zhou Dexiang for their help with control related jargons and latex programming respectively.

Last but not least, I would like to thank my wife Fangfang for her unwaivering love, patience, encouragement and company throughout the course of my study. Special mention to my parents and sister for their continuous encouragement and support on my decision for further studies.

Abstract

Systolic Array architectures are data-flow based but designing architectures for solving specific problems can pose a challenge. In this thesis, an investigation into a scalable design for accelerating the problem of solving a dense linear system of equations using LU Decomposition is presented. A novel systolic array architecture that can be used as a building block in scientific applications is described and prototyped on a Xilinx Virtex 6 FPGA. The proposed linear solver has a throughput of approximately 1 million linear systems per second for matrices of size $N = 4$ and approximately 82 thousand linear systems per second for matrices of size $N = 16$. In comparison with similar work, the proposed design offers up to a 12x improvement in speed whilst requiring up to 50% fewer hardware resources. As a result, a linear system of size $N = 64$ can now be implemented on a single FPGA, whereas previous work was limited to $N = 12$ and resorted to complex multi-FPGA architectures to achieve the same effect. Moreover, the scalable design can be adapted to different sized problems with minimum effort.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ASIC	Application Specific Integrated Circuits
ASM	Active Set Method
AST	Altera Synthesis Tool
BLAS	Algebra Subprograms
BRAM	Block RAM
BSV	Bluespec SystemVerilog
CGM	Conjugate Gradient Method
CORDIC	COordinate Rotation DIgital Computer
CUDA	Compute Unified Device Architecture
DSP	Digital Signal Processing
DSPB	DSP Builder
EDK	Embedded Development Kit
FGM	Fast Gradient Method
FIR	Finite Impulse Response
FPU	Floating Point Unit
FRADL	FPGA Regular Array Description Language
GPP	General Purpose Processors
GPU	Graphics Processor Unit
FPGA	Field Programmable Gate Array
HDL	Hardware Descriptive Language
HLS	High-Level Synthesis
IPM	Interior Point Method
KKT	Karush-Kuhn-Tucker

LAPACK	Linear Algebra PACKage
LNS	Logarithmic Number System
LUD	LU Decomposition
MAC	Multiply-Add-Subtract
MIMD	Multiple-Instruction-Multiple-Data
MIMO	Multiple Inputs Multiple Outputs
MINRES	Minimum Residual
MMS	Modified Multiply-Subtract
MPC	Model Predictive Control
NI	National Instruments
NRE	Non-recurring Engineering
OpenCL	Open Computing Language
PE	Processing Elements
QP	Quadratic Programming
QRD	QR Decomposition
QRD-RLS	QR Decomposition-based Recursive Least Squares
RHS	Righ-hand-side
RTL	Register-Transfer-Level
SA	Systolic Array
SDR	Software Defined Radio
SIMD	Single-Instruction-Multiple-Data
SVD	Singular Value Decomposition
SysGen	System Generator
TI	Texas Instruments
TSA	Triangular Systolic Array
TTM	Time-to-Market
VCD	Value Change Dump
VLIW	Very Long Instruction Word
WLF	Wave Log Format

XST	Xilinx Synthesis Tool

Chapter 1

Introduction

1.1 Linear Solver

A large portion of scientific computing is concerned with solving a system of linear equations through the use of numerical methods to give approximate but accurate solutions to computationally complex problems. Solving a system of linear equations is the basis of a number of scientific applications and two approaches are often used: iterative and direct methods. Iterative methods generate a sequence of approximations to the solution and the same computation procedure is performed in a repeated manner. Although iterative methods can be efficient both computationally and in terms of storage, a very good initial approximate value has to be chosen. This is because iterative methods are prone to numerical inaccuracies and convergence issues and the time taken to compute the exact solution becomes unpredictable. In contrast, direct methods determine the exact solution through a finite sequence of operations. As a result, the exact solution can be computed in a predictable amount of time and storage requirements can be estimated at design time.

1.2 Scientific Computing Platforms

As with all algorithmic implementations, development engineers usually survey a wide-range of mainstream computing platforms in the ever-changing landscape of computing architecture. The main reason for this change is due to the scaling limitations of clock frequency as chip manufacturers innovate to grapple with minimizing the amount of leakage current with each shrink in die size. More recently, semiconductor manufacturers have looked to increasing the amount of parallelism, and this has brought about the era of multi-cores with the latest General Purpose Processors (GPP), such as the Intel i7-3960X [7], featuring 6 cores, with each core operating at 3.3GHz and a combined onboard cache size of 15Mbytes. As general as a GPP is, real-world performance is usually much lower than the theoretical peak performance due to the fixed datapath and general architecture of the GPP computing platform. It is only recently that the software industry has caught up with chip manufacturers to exploit the computation power of multiple processor cores. In the past decade, two other scientific computing platforms have gained prominence for their alternative architectures, the Graphics Processor Unit (GPU) and the Field-Programmable Gate Array (FPGA), offering data parallelism to achieve significant hardware acceleration.

Modern day GPUs, such as those from AMD/ATI [8], contain hundreds of processor cores that can perform specialized matrix computation in a massively parallel manner, using the Very Long Instruction Word (VLIW) computing architecture. These devices boast high peak theoretical performance for single precision floating point [9], of the order of >1 TFLOP/s. But such performance is only achievable if sufficient parallelism can be applied, with some thousands of parallel threads, and provided there are no race conditions for read/write operations in the memory sub-system.

A different approach can be undertaken through the exploration of custom computing architecture. Application Specific Integrated Circuits (ASICs) are known for having very high computation performance, low power consumption and offer a small die size. But the complexity of designing and validating such ASIC devices become a barrier as systems grow in design complexity and experienced hardware designers become scarcer. ASICs are known to offer long time-to-market (TTM), require high upfront costs and are only suitable for large volume applications. On the contrary, innovative advances in reconfigurable computing have made FPGAs a suitable platform for accelerating scientific computations. FPGAs boast fast TTM, low upfront costs and design errors can be rectified easily in the field unlike ASICs.

Recently, low to mid-range FPGAs have been embedded with dedicated hardware resources; Digital Signal Processing (DSP) blocks are included on-chip, equipping FPGAs with more computational muscle whilst providing deterministic execution time and low power consumption. With the availability of on-chip DSP blocks, most of the general FPGA fabric can be available for other hardware tasks. The highly optimized and onboard DSP blocks help accelerate the computational performance of DSP intensive algorithms. Given knowledge of the target FPGA architecture and embedded resources, hardware designers can better optimize their hardware design to exploit available embedded resources. A recent FPGA, Xilinx Zynq 7000 [10], includes an ARM dual-core Cortex A9 MPCore microprocessor, residing on the same silicon die as the general-purpose FPGA fabric. An implicit benefit of residing on the same silicon die is a high performance coupling of the two components.

On the design front, the availability of IP cores for common hardware peripherals seamlessly adds to the popularity of FPGAs. In addition, the productivity of hardware designers is increased and more effort can be spent in designing complex systems. A key disadvantage of FPGAs is design complexity compared to GPPs. The amount of onboard FPGA fabric required to synthesize the required hardware logic is also constrained by the efficiency of the design decisions made by the respective synthesis software tools.

FPGAs have previously been used to accelerate Model Predictive Control (MPC) and offer the benefit of hardware accelerated performance, but with the flexibility to tailor the implementation to the specific problem of interest. The majority of MPC researchers have focused on solving large problem sizes and their system solvers utilize iterative algorithms, such as the Conjugate Gradient Method, to handle sparse matrices. To achieve high linear solver performance, a digital hardware designer must customize the design at a low level of detail. But the design complexity in utilizing such linear solvers is well-beyond the reach of non-circuit designers and scientific researchers in general. Moreover, the ability to connect algorithms to hardware architectures and the use of high-level software tools for rapid design prototyping and parameterization is not exploited.

1.3 High-Level Design Approaches

The popularity of FPGAs, in realizing application-specific or hardware accelerator systems, is traditionally attained through the use of Hardware Descriptive Languages (HDL) such as VHDL and Verilog. In comparison with traditional high-level programming languages, the HDL-based design methodology provides a relatively low level of abstraction [11]. The pre-requisite for HDL design is prior/existing background knowledge and experience in digital design techniques, such as Register-Transfer-Level (RTL), in order to exploit the underlying FPGA architecture. Moreover, designing and troubleshooting of the application-specific design alone is time-consuming and this is why the use of FPGAs is limited to digital hardware design experts. The design complexity of state-of-the-art systems drives a strong need to have a technology independent modelling tool to model such complex systems while reducing the design effort required before the first prototype ASIC chip is taped-out to an IC package. Note that the cost of developing the ASIC chip amounts to millions of dollars and any design changes will further marginalize the companys profits per chip.

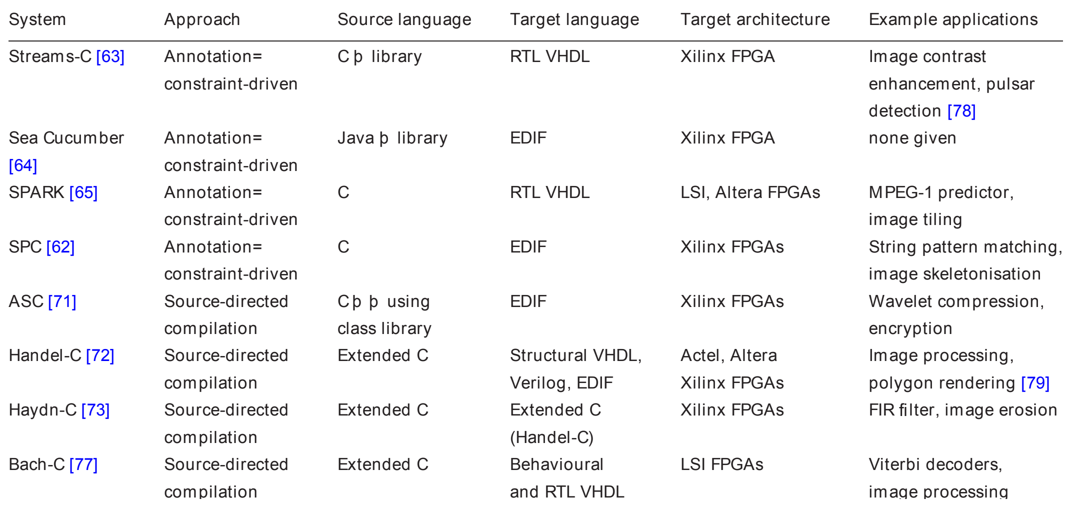
High-Level Synthesis (HLS) and compilers were created to promote the widespread use of FPGAs for both digital and non-digital hardware design specialists. The aim of HLSs is to allow the designers/programmers to rapidly create and model the complex systems before generation of RTL design code. For example, the HLS tools allow the designer to create a digital circuit, such as an embedded controller, with ease on an FPGA through the use of high level languages such as C, MATLAB and LabVIEW [12]. Hence, overall productivity of the designer/programmer is increased and more time can be spent on verifying the functional correctness of their customized system design. Examples of popular HLS tools include Handel-C [13], ImpulseC [14], Xilinx Embedded Development Kit (EDK) [15], MATLAB Simulink HDL Coder [16], National Instruments (NI) FPGA [12], Synphony [17], Vivado [18] and Bluespec [6]. The proposed approaches are discussed in the following sub-sections.

1.3.1 Source-directed compilation approach

[1] surveyed the architecture and design methods aspect of reconfigurable computing for various applications. Various design approaches including general-purpose, special purpose, other design methods and emerging directions were reported. For general-purpose methods, [1] shortlisted the more significant hardware compilers and illustrated their corresponding use of various source and target languages, as shown in Figure 1.1.

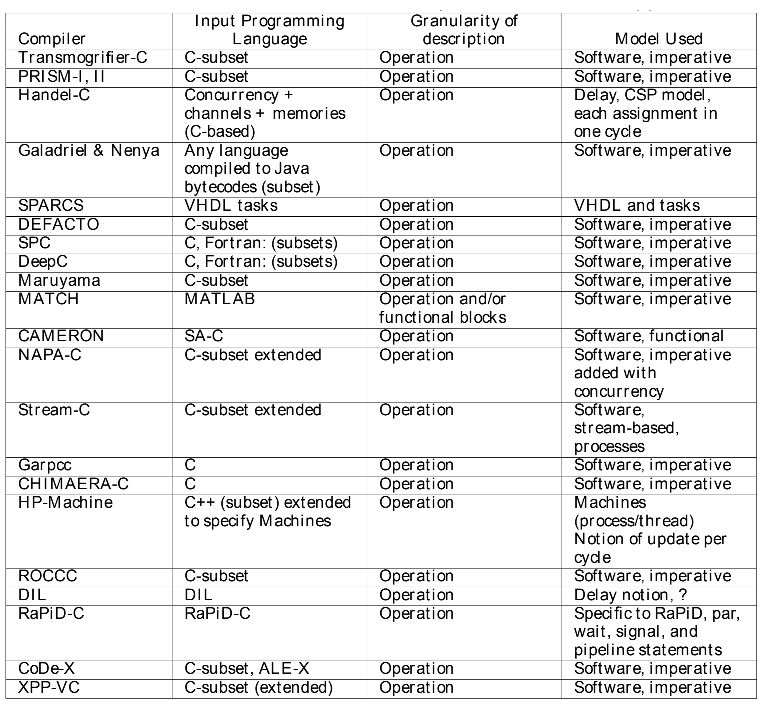
Similarly in 2010, a comprehensive survey of C-to-FPGA tools was conducted by [2] and part of the survey focused on identifying the compilation method and the synthesis techniques applied, see Figure 1.2.

The findings in both [1, 2] suggest that C language is the preferred source language and the potential benefits of such an approach is well understood. To exploit the target FPGA hardware performance and resources efficiently, variants of C language are proposed to enhance the expressiveness of the C language for FPGAs. In addition, FPGA platform architectures vary for different manufacturers. As a result, more research efforts are required to make the source-directed compilation



System	Approach	Source language	Target language	Target architecture	Example applications
Streams-C [63]	Annotation= constraint-driven	C þ library	RTL VHDL	Xilinx FPGA	Image contrast enhancement, pulsar detection [78]
Sea Cucumber [64]	Annotation= constraint-driven	Java þ library	EDIF	Xilinx FPGA	none given
SPARK [65]	Annotation= constraint-driven	C	RTL VHDL	LSI, Altera FPGAs	MPEG-1 predictor, image tiling
SPC [62]	Annotation= constraint-driven	C	EDIF	Xilinx FPGAs	String pattern matching, image skeletonisation
ASC [71]	Source-directed compilation	C þ þ using class library	EDIF	Xilinx FPGAs	Wavelet compression, encryption
Handel-C [72]	Source-directed compilation	Extended C	Structural VHDL, Verilog, EDIF	Actel, Altera Xilinx FPGAs	Image processing, polygon rendering [79]
Haydn-C [73]	Source-directed compilation	Extended C	Extended C (Handel-C)	Xilinx FPGAs	FIR filter, image erosion
Bach-C [77]	Source-directed compilation	Extended C	Behavioural and RTL VHDL	LSI FPGAs	Viterbi decoders, image processing

Figure 1.1: General-purpose Hardware Compilers [1]



Compiler	Input Programming Language	Granularity of description	Model Used
Transmogrifier-C	C-subset	Operation	Software, imperative
PRISM-I, II	C-subset	Operation	Software, imperative
Handel-C	Concurrency + channels + memories (C-based)	Operation	Delay, CSP model, each assignment in one cycle
Galadriel & Nenya	Any language compiled to Java bytecodes (subset)	Operation	Software, imperative
SPARCS	VHDL tasks	Operation	VHDL and tasks
DEFACTO	C-subset	Operation	Software, imperative
SPC	C, Fortran: (subsets)	Operation	Software, imperative
DeepC	C, Fortran: (subsets)	Operation	Software, imperative
Maruyama	C-subset	Operation	Software, imperative
MATCH	MATLAB	Operation and/or functional blocks	Software, imperative
CAMERON	SA-C	Operation	Software, functional
NAPA-C	C-subset extended	Operation	Software, imperative added with concurrency
Stream-C	C-subset extended	Operation	Software, stream-based, processes
Garpcc	C	Operation	Software, imperative
CHIMAERA-C	C	Operation	Software, imperative
HP-Machine	C++ (subset) extended to specify Machines	Operation	Machines (process/thread) Notion of update per cycle
ROCCC	C-subset	Operation	Software, imperative
DIL	DIL	Operation	Delay notion, ?
RaPiD-C	RaPiD-C	Operation	Specific to RaPiD, par, wait, signal, and pipeline statements
CoDe-X	C-subset, ALE-X	Operation	Software, imperative
XPP-VC	C-subset (extended)	Operation	Software, imperative

Figure 1.2: Main Characteristics of various HLL C-to-FPGA Compilers [2]

approaches sufficiently attractive for low-level circuit designers to adopt.

1.3.2 Digital Signal Processing IP Cores

A typical drawback of using such high-level tools lies in the translation overheads, which result in non-optimum performance and inefficient use of the FPGAs on-chip resources, such as the inability to parallelize the hardware design sufficiently to gain higher system throughput, as seen in [19]. The availability of IP cores attempts to address such problems and is made available through software add-ons such as Xilinx Core Generator [20], Altera MegaCore Functions [21] and OpenCores [22], with an aim to reduce the development cycle to productively build modern day complex systems. To integrate and debug the IP cores with the existing logic, digital hardware designers require to be in the loop. The main disadvantage of using IP cores is the reliability of the IP core as bug fixes can only be issued by respective FPGA/CAD vendors and this introduces delay to the digital hardware designers already tight project schedule. As a result, architecture exploration becomes prohibitive when vendor-protected IP cores are involved. Similarly, OpenCores IPs are user-contributed and unoptimized with the option for user-modification to be made through HDL.

From the constraints mentioned, a model-based approach is more appropriate as the programming environment is graphical and data visualization becomes natural for non-circuit hardware designers.

1.3.3 Model-based Approach

The model-based approach utilizes graphical environments, such as MATLAB Simulink environment for design prototyping and verification of DSP applications on FPGAs. Once the functional hardware design have been verified, the downstream FPGA implementation steps involving synthesis and Place-and-Route are automatically performed to generate an FPGA programming file. The two popular offerings are Xilinx System Generator (SysGen) [23] and Altera DSP Builder (DSPB) [24]. Compared with previous approaches, the model-based approach enables rapid creation of custom peripherals when compared to programmatic flow using HDL and previous experience with FPGAs is not required. In this thesis, the use of SysGen is selected for design prototyping and verification of a data-flow based hardware architecture for linear solvers.

1.4 Research Goals and Contributions

In this thesis, we aim to propose a scalable and parameterizable linear solver as a building block in scientific applications, such as MPC. The structural regularity and scalability of the systolic array approach allows the linear solver to be rapidly prototyped using high-level software tools and non-circuit designers only need work at the architecture level. The proposed scalable systolic array architecture is not constrained to MPC and can be applied to general scientific computing problems where a system of linear equations is to be solved.

The major contributions of the thesis are as follows:

1. Exploitation of architectural parallelism, idle sequential cycles and omission of redundant arithmetic operations resulted in a novel systolic array hardware design architecture.
2. Scalable and wordlength parameterizable hardware architecture for easy adaptation to different sized problems with minimum effort.
3. Proposed linear solver performs floating-point division and has a throughput of about 1 million linear systems for matrices of size $N = 4$ and about 82 thousand linear systems for matrices of size $N = 16$.
4. Proposed design offers up to 12x improvement in solver speed whilst requiring up to 50% less hardware resources when compared to similar works.

5. A large linear solver of size $N = 64$ can now be implemented on a single FPGA chip, whereas previous work was limited to $N = 12$ and resorted to complex multi-FPGA architectures to achieve the same effect.

1.4.1 Publications

A conference poster [25] has been accepted and presented in IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP) 2014.

1.5 Thesis Organization

This thesis is organized as follows. Chapter 2 presents a background on how solving a system of linear equations will accelerate scientific applications, such as Model Predictive Control (MPC) and presents the various design approaches to accelerate the linear solver. Chapter 3 summarizes prior work using general scientific platforms and FPGAs as a linear solver hardware accelerator. Chapter 4 presents a case-study to access the performance and hardware trade-off between Bluespec and MATLAB Xilinx SysGen tools using a direct-form FIR filter hardware design. Chapter 5 describes the proposed hardware architecture to enable design scalability. Chapter 6 presents the implementation setup and discusses the results in comparison with similar work. Chapter 7 describes the thesis' conclusion and highlights the outstanding issues for implementation as future work.

Chapter 2

Background

In this chapter, relevant background material is described. Firstly, the motivation for accelerating the system of linear equations in Model Predictive Control application is described. Secondly, an introduction to systolic arrays and existing work on LU Decomposition based linear solvers is presented. Thirdly, various high-level design approaches are reviewed. Finally, a brief overview of Bluespec and Xilinx System Generator tools are described for reader's reference prior to the case study experiment in Chapter 4.

2.1 Model Predictive Control

Model Predictive Control (MPC) is an advanced control method that is well established in the petrochemical industry. The natural ability in handling large multiple inputs multiple outputs (MIMO) systems with physical constraints makes MPC attractive. Characteristics of MPC include the moving horizon implementation, performance oriented time domain formulation, incorporation of constraints and explicit system model for use in predicting the future plant dynamics [3]. Typical components of MPC include the prediction model, objective function and obtaining the control law, as shown in Figure 2.1. Conventional MPC requires that the sampling interval be greater than the time taken to solve the optimization problem [26] as it uses a model of the system to be controlled, to solve using numerical optimization methods. As the first part of the solution is implemented, the deviation error will occur between the next output measurement and the controller′s predicted trajectory. Thus, the optimal control problem is only updated with new data after new measurement data has to be obtained at the next sample instant. This process is repeated for future sample instances.

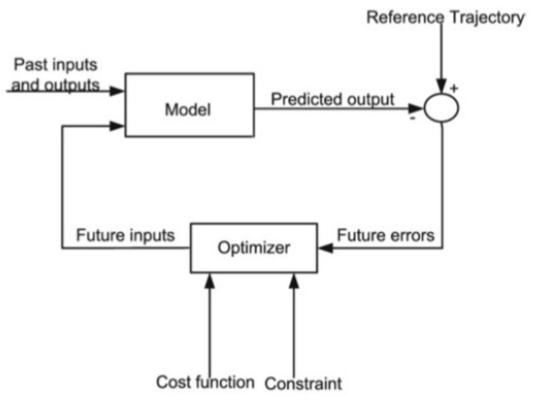


Figure 2.1: Basic Structure of MPC [3]

The criteria for real-time MPC strongly depend on the speed at which the optimization problem is solved, in order to control the plant or system more quickly and effectively. In addition, the optimization problem requires sampling at high frequencies in order to capture fast occurring disturbances. That is why current applications of MPC are restricted to slow processes such as chemical plants, with sampling periods in the order of seconds although [27] reports growing research interest into the use of MPC in other areas such as ships, aerospace and micro scale devices. Therefore, it is useful to briefly understand the fundamental concepts on how Constrained MPC can be formulated as a Quadratic Programming (QP) problem.

Assume a discrete linear time-invariant plant with the following state space form:

$$\sum \begin{cases} x(k+1) & = Ax(k) + Bu(k) \\ y(k) & = Cx(k) \end{cases} \tag{2.1}$$

where $y(k) \in \mathbb{R}^p$,$u(k) \in \mathbb{R}^m$ and $x(k) \in \mathbb{R}^n$ represent the system output, input and internal states, respectively. The constrained MPC problem's objective is to minimize the cost function of:

$$\Phi\left(y, \triangle u\right) = \sum_{j=1}^{N_p} \|y(k+j) - \omega(k+j)\|_q^2 + \sum_{j=0}^{N_{u-1}} \|\triangle u(k+j)\|_r^2$$

where the cost function is subject to the following inequality constraints,

$$y_{LB} \leq J_y y \leq y_{UB} \tag{2.2}$$

$$x_{LB} \leq J_x x \leq x_{UB} \tag{2.3}$$

$$u_{LB} \leq J_u u \leq u_{UB} \tag{2.4}$$

$$\hat{u}_{LB} \leq J_{\hat{u}} \triangle u \leq \hat{u}_{UB} \tag{2.5}$$

The MPC problem can be converted into a QP problem [27] using the standardized approach by first replacing the predicted system output with the following definitions:

$$\Psi_u = \begin{bmatrix} CB & 0 & \cdots & 0 \\ CAB & CB & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ CA^{N_P-1}B & CA^{N_P-2}B & \cdots & CA^{N_P-N_u}B \end{bmatrix}$$

$$\Psi_u' = \begin{bmatrix} CA \\ CA^2 \\ \vdots \\ CA^{N_P} \end{bmatrix} z = \begin{bmatrix} \triangle u(k) \\ \triangle u(k+1) \\ \vdots \\ \triangle u(k+N_u-1) \end{bmatrix}$$

Hence, the constrained MPC problem can be formulated as a compact QP problem:

$$\underset{z \in R^{n_v}}{min} \left\{ \frac{1}{2} z'Qz + c'z \; : \; Jz \leq g \right\} \tag{2.6}$$

From equation (2.6), $n_v = mN_u$ denotes the number of decision variables while $Q = 2\Psi'_u\Psi_u{+}2I$ is an $n_v{\times}N_u$ positive definite Hessian matrix and $c = 2\Psi'_u\left(\Psi_x x_k - \omega\right)$ is an $n_v \times 1$ column vector. In addition, the size of J and g are $m_c \times n_v$ matrix and $m_c \times 1$ respectively, where m_c represents the total number of inequality constraints on z. Here, ω is the set-point while N_p and N_u are the prediction and control horizons respectively.

At each sampling instance, the QP solver will generate the corresponding control signal to the plant. At the next sampling instance, the optimization solver process is repeated. The size and complexity of the optimization problem affects the computational requirements for online computation, condemning the use of MPC further. That is why the mentioned approach can be replaced with that proposed by Wright [28](pp. 91 of [29]), where the states and inputs during the prediction horizon are kept as variables, in order to get a banded Q matrix. The consequent solution of the linear system of equations, in both Interior Point Method (IPM) and Active Set Method (ASM), can be obtained more quickly for problems with large sizes. In the proposed research, we assume the MPC problems are insufficiently large to adopt the proposed QP formulation approach in [28].

Common methods for solving the QP problems are the ASM and IPM. In this thesis, IPM [30] is assumed and the Karush-Kuhn-Tucker (KKT) conditions for optimality is applied to the infeasible IPM for solving a QP problem. Thereafter, optimal control can be computed by applying Newton's method and the search direction can be obtained.

From Algorithm 1, it can be seen that step 3 is the most computationally intensive, which solves a system of linear equations, Ax=$\hat{b}$. Numerical methods such as Cholesky, Minimum Residual (MINRES), Conjugate Gradient Method (CGM),

Algorithm 1 Interior Point Method

```
1: Start IPM and select initial conditions
2: procedure INITIALIZE $(z^0, \lambda^0, t^0)$ WITH($(\lambda^0, t^0)>0$)
3:     At k-th iteration, solve for $\left(z^k, \lambda^k, t^k\right)$ with:
       $\begin{bmatrix} Q & J' \\ J & \Gamma \end{bmatrix} \begin{bmatrix} \Delta z^k \\ \Delta\lambda^k \end{bmatrix} = \begin{bmatrix} r_1^k \\ r_2^k \end{bmatrix}$ where $\Gamma = -\left(\Lambda^k\right)^{-1} T^k$
4:     Increment variables and check for convergence
5:     if Converge, then
6:         Stop and obtain optimal control $z^{k+1}$
7:     else
8:         Return to step 3 and continue iteration
9:     end if
10: end procedure
```

Singular Value Decomposition (SVD), QR Decomposition (QRD), and LU Decomposition (LUD) are some of the approaches used for linear solvers.

In this thesis, we will focus on developing a linear solver for general use. Direct methods are preferred so as to enable efficient mapping to FPGAs and special matrix properties, such as symmetry and positive definite, are not considered. Secondly, the constraints and A matrix, in the compact QP problem, are assumed to be represented in a dense matrix data structure and the MPC problems are insufficiently large to adopt the proposed QP formulation approach [30] with high linear solver speed requirements, $\geq$1000 linear systems per second. One aspect of our proposed research is focused on utilizing an efficient and easy to implement numerical method for the linear solver. A single right-hand-side (RHS) is assumed and LU Decomposition is preferred as the method can be easily mapped to systolic arrays. To solve for a system of linear equations of size N, three steps are performed:

1. LU Decomposition A = LU
2. Forward Substitution L$\hat{y} = \hat{b}$
3. Backward Substitution U$\hat{x} = \hat{y}$

The computational complexity for LU Decomposition is $\mathcal{O}\left(N^3\right)$ and requires division operations. The options for replacing the division operations are COordinate

Rotation DIgital Computer (CORDIC) [31], Givens Rotation and Conjugate Gradient Method to achieve division free operations. In this thesis, floating-point division is performed and the proposed use of Systolic Array architecture reduces computational complexity to $\mathcal{O}(N)$. In support of our systolic array approach, implementations of LU-based linear solver will be presented and discussed in Chapter 6.

2.2 Systolic Arrays

Parallel execution of a problem can accelerate scientific computations with applications ranging from simulations to data mining. The common objective is to reduce time taken to solve large scientific problems by placing more emphasis on exploiting multi-processor hardware. If the single processing unit can accomplish the task in time T then N processing units could ideally accomplish it in time T$/N$, termed linear speed-up. But in most cases, the actual speed-up achieved on parallel computers is considerably smaller than the desired linear speed-up and this can be explained using Amdahl′s law. Amdahl's law is often used to predict the theoretical maximum speedup for program processing using multiple processors. The maximum achievable speedup using N number of processors is $\frac{1}{(1-P)+\left(\frac{P}{N}\right)}$, where P is the proportion of a system or program that can be made parallel. Taxonomies to expose parallel architectures from different viewpoints have been reported by Flynn and Duncan [5].

Systolic and wavefront arrays are also known as VLSI processor arrays and their differences are contrasted in Figure 1.1 along with Single-Instruction-Multiple-Data (SIMD) and Multiple-Instruction-Multiple-Data (MIMD) architectures. Systolic Arrays (SAs) are generally classified as high-performance, special-purpose VLSI computer systems that are suitable for specific application requirements that must balance intensive computations with demanding I/O bandwidths. SAs are organized as networks comprising a large number of identical, locally connected elementary Processing Elements (PE). A global clock synchronizes data between the PEs. Data in SA are rhythmically pulsed from memory through PEs before returning to mem-

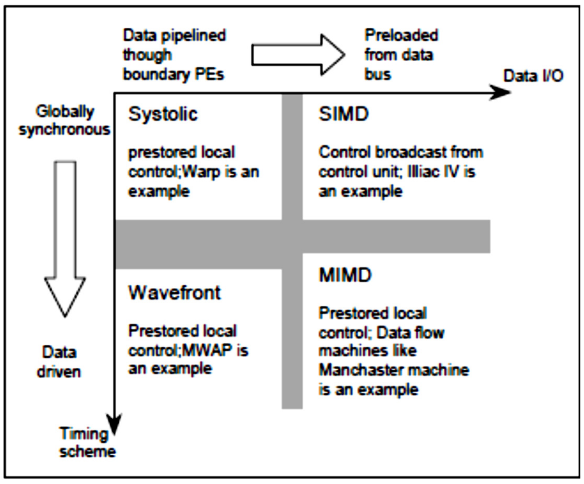

Data pipelined though boundary PEs
Preloaded from data bus
Data I/O
Globally synchronous
Systolic
prestored local control;Warp is an example
SIMD
Control broadcast from control unit; Illiac IV is an example
Wavefront
Prestored local control;MWAP is an example
MIMD
Prestored local control; Data flow machines like Manchaster machine is an example
Data driven
Timing scheme

Figure 2.2: Similarities and differences of architectures (SIMD, MIMD, Systolic Arrays and Wavefront Arrays) [4] [5]

ory. Hence, only the initial data and final results are transferred between the host and the systolic array.

A computing network may be considered a SA provided it exhibits the following characteristics: Network, Rhythm, Synchrony, Modularity, Regularity, Locality (Spatial/Temporal), Boundary, Extensibility and Pipelinebility; Network and rhythm have been mentioned previously. Synchrony refers to the behaviour of data that is rhythmically computed, by global clock, and passed through the network for execution of instructions and communication purposes. Modularity refers to the finite/infinite array consisting of modular processing units. Regularity refers to a homogeneously interconnected modular processing unit. Spatial locality refers to the local communication between cells. Temporal locality refers to the signal transmission characteristic of at least one unit time delay between cells. Boundary processors are the PEs are the only ones allowed to communicate with the outside world. Pipelinebility is a synonym for data pipelining where at least one delay element is found or inserted between two directly connected combinatorial PEs for the purpose of achieving high speed.

The concept of SAs to achieve data parallelism was introduced by [32] and global communication was an issue for large hardware designs. [32] noted that mathe-

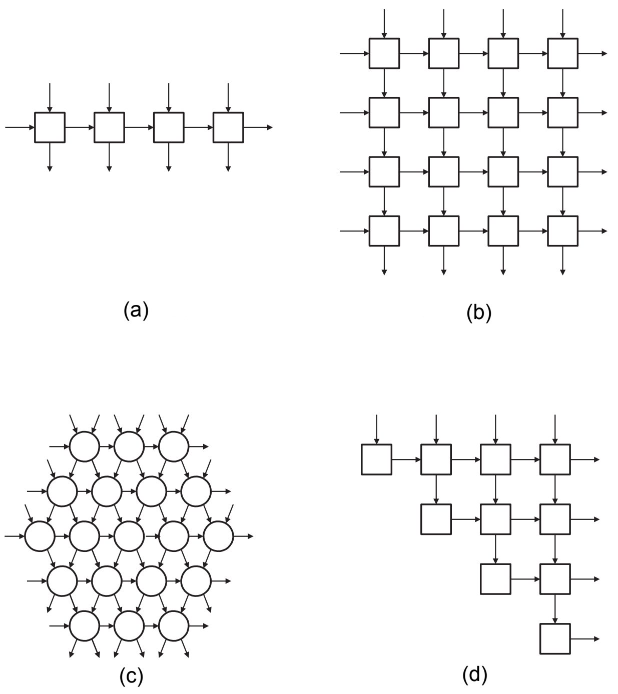
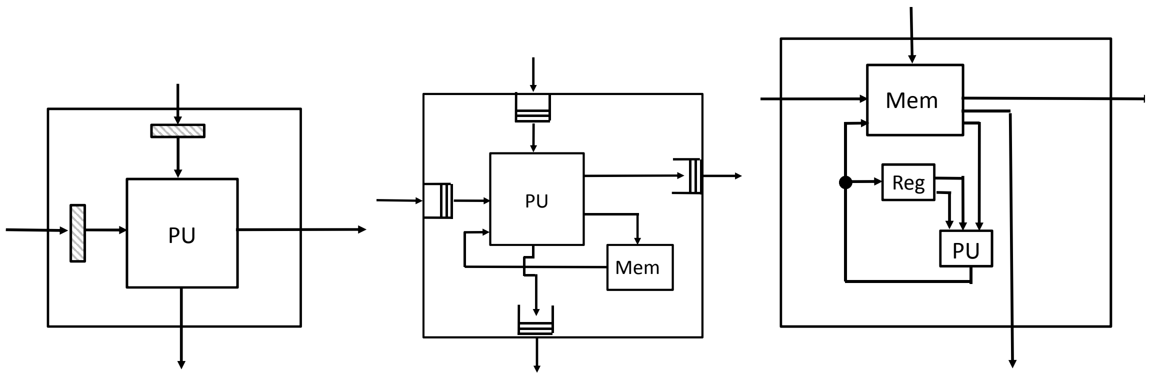


Figure 2.3: (a) Linear (b) Orthogonal (c) Hexagonal (d) Triangular [5]

matical pivoting was more for storage minimization rather than global communication and subsequently proposed two triangularization concepts, triangularization with neighbour pivoting and orthogonal triangularization, with an aim of minimizing global communication. Some examples of SA structures are linear arrays, orthogonal, hexagonal and triangular structures which can be implemented in a variety of hardware technologies, see Figure 2.3. Despite being proposed three decades ago, SAs are still an important research area for high performance computing applications [33]. Some applications of SAs include Multiple Input Multiple Output (MIMO) Software Defined Radio (SDR) [34], Block Matching Motion Estimation [35], and QR Decomposition for radar mapping applications [36]. The homogeneous multiprocessor array permits matrix computations naturally and enables high-level computations to be easily mapped into hardware structures through regular and reconfigurable pipelined processor cells.

In relation to communication bandwidth and internal storage, three classes of systolic-type arrays exists, namely systolic cell, pseudo-systolic cell and local-access



(a) Systolic **(b)** Pseudo-systolic **(c)** Local-access

Figure 2.4: Classes of cells in systolic-type arrays

cell. In this thesis, the author assumes off-chip memory degrades the overall system performance and on-board memory is to be exploited on the resource-constrained FPGA platform. Hence, the *pseudo-systolic cell* type of SA is selected, see Figure 2.4. Readers are referred to [37] for more details of each class of systolic-type array.

Although SAs are organized as networks comprising of a large number of identical, locally connected elementary Processing Elements (PE), a global clock is required to synchronize data between the PEs. Data in the SA is rhythmically pulsed from memory through PEs before returning to memory. Such synchronous behaviour may not be needed on all applications and can cost resources. Hence, the use of modern high-level software tools is proposed to reduce the time-to-market and non-recurring engineering costs for implementation on low-cost FPGA platforms.

The simple design of the PE suggests that SA design is easily scalable and can achieve high data throughput. In cases where design size is limited by available hardware resources, SAs may also be configured to operate as a high performance co-processor to accelerate numerical calculations and communication with an external controller is performed through a high speed interconnect bus, see Figure 2.5. A case in point is where triangular systolic array for the matrix triangularization step is proposed to reduce computational complexity to $\mathcal{O}(N)$. The reader is assumed to

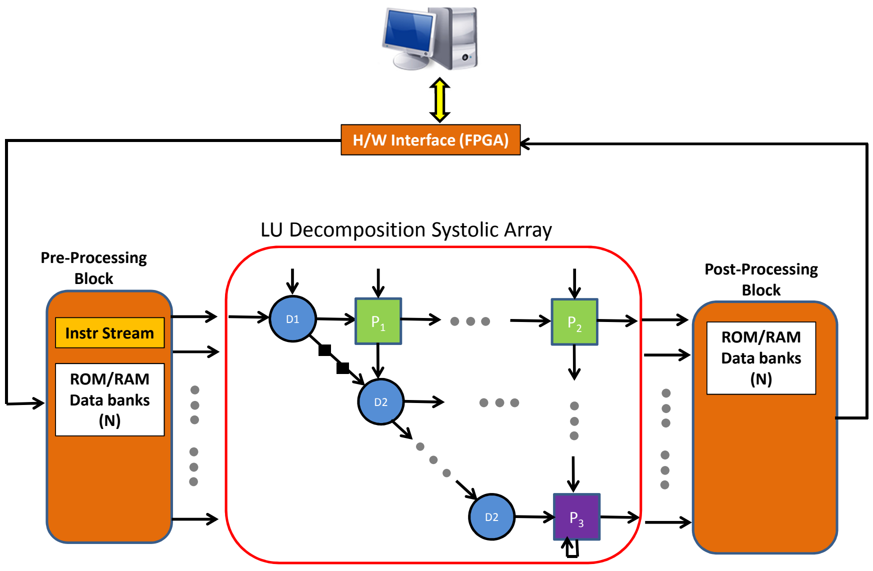


Figure 2.5: Overall System Architecture

be familiar with matrix decomposition methods and LU decomposition is preferred as the method is easily mapped to systolic arrays.

2.3 Software Linear System Solvers

Software libraries, such as Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK), exist to solve systems of linear equations on GPPs and GPUs. While there are many published research works on BLAS and LAPACK for FPGAs [38], the libraries are highly customized and require circuit hardware designers to be in the loop. To realize our research objective of a scalable linear solver for non-circuit designers, the use of high level synthesis tools, such as Bluespec and Xilinx System Generator, are highly desirable and are briefly examined in this chapter. To determine the suitable high-level software environment, a case study was performed to compare and contrast both synthesis tools and performance results are discussed in details in Chapter 4.

2.3.1 Bluespec

Bluespec [39] is a state-of-the-art platform tool that is utilized for the purpose of hardware system design specification, synthesis, modeling and verification [40]. To date, Bluespec has been used in the area of modeling for software development, modeling for architecture exploration, verification and IP creation. As with other HDLs, Bluespec designs are modular and each Bluespec module communicates through an interface, instead of ports. A module is used to represent hardware circuits in Bluespec, similar to Verilog module. Each module consists of 3 basic aspects namely states, rules and interfaces, as shown in Figure 2.6.

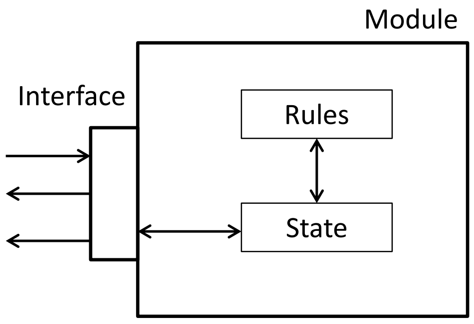


Figure 2.6: Bluespec Design Flow [6]

The states are represented as hardware registers, flip-flops and memories. Rules are used to execute operation logic to satisfy/modify the state values and each module can contain multiple rules. Each interface consists of methods, akin to methods in object-oriented programming (OOP) languages, and inherits the advantage of OOP for large system designs. For communication between interfaces, Bluespec has implicit guards on the methods, which specify when a method is ready for firing.

The language, Bluespec SystemVerilog (BSV), is based on Haskell and the rule-based approach, known as Guarded Atomic Action, is used to describe hardware behaviour. BSV programs can be understood in terms of atomic rule firings in the hardware design and traditional hardware model of finite state machine is used to implicitly express concurrent operations. In instances where multiple rules are concurrently executed, the Bluespec compiler is able to generate a combinational

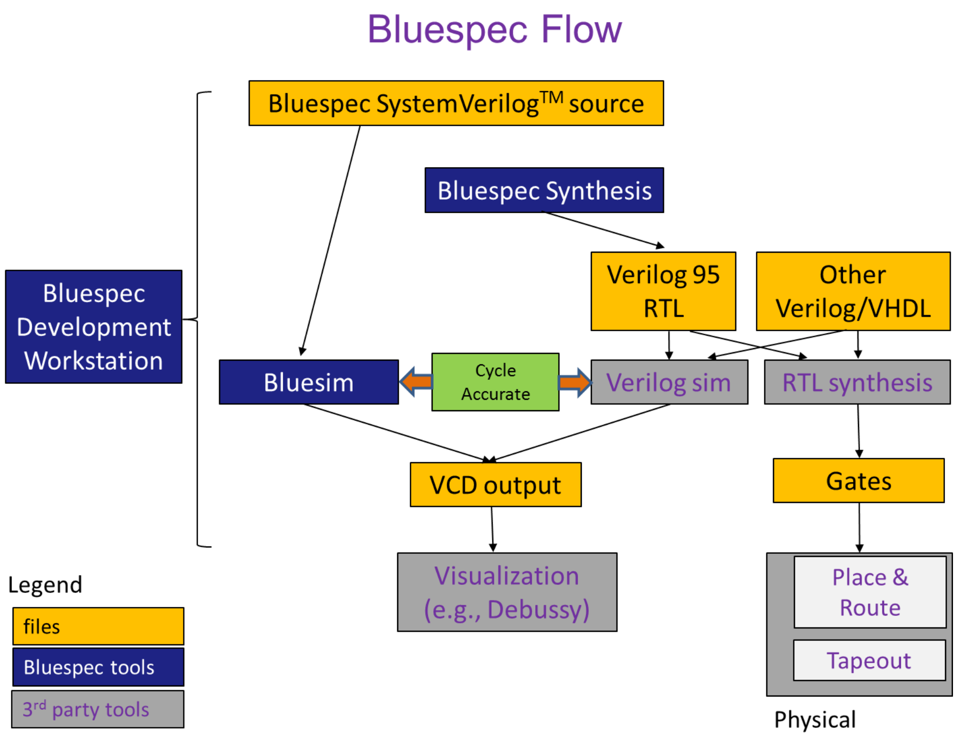
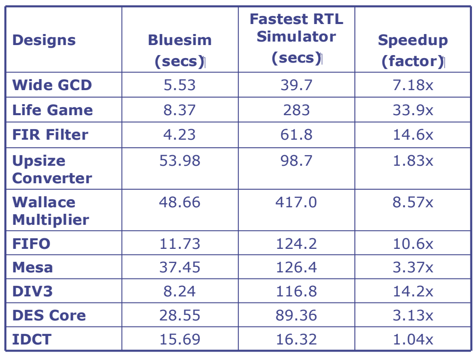


Figure 2.7: Bluespec Software Module [6]

scheduler, which efficiently schedules the rules in each cycle. In the event where the Bluespec compiler is unable to decide, feedback will be provided to notify and guide the hardware designer in making better design decisions. For example, if the Bluespec compiler determines that two rules cannot be concurrently fired and additional combinational logic is required, the designer will have to explicitly specify his preference to the compiler. Thereafter, the compiler is able to work through all rules and apply Boolean optimizations to simplify the hardware design further resulting in the generation of a complete schedule. The rule-based design is compiled and translated into RTL implementation, as shown in Figure 2.7. Likewise BSV has in-built support for importing Verilog IP and C code into BSV designs through the use of wrappers.

Bluespec is strongly type-checked to reduce the amount of logic bugs, a feature commonly found in traditional Hardware Descriptive Language (HDL). For easier understanding of synthesizable hardware, state elements have to be explicitly created. According to [41], the BSV-coded Verilog can approach the quality of hand-coded Verilog design as supported by the work described in [42] where a comparative evaluation of a Reed-Solomon Decoder was performed between a C-based design flow and Bluespec. [42] pointed out that the advantage of a C-based synthesis tools compiler decreases as data-dependant control behaviour in the program increases, leading to inefficient hardware. Even in the hands of an experienced hardware designer, considerable effort is still required for architecture design exploration. On the other hand, [42] acknowledged that while Bluespec simplifies the algorithm into relatively simple modular structures, micro-architecture exploration is still required on a per application basis. Bluespec architectural exploration can be expected in the proposed research work.

Additional facilities of Bluespec include Bluesim, a native simulator and source-level debugger, touted to provide 5-50x speed improvements over Verilog simulation through exploitation of the computation model. A table of comparison between Bluesim and a commercial RTL simulator is presented in Figure 2.8. Bluesim has in-built debugging options to enable/disable viewing of scheduling activities and generates 2 signals, "CAN_FIRE_rulename" and "WILL_FIRE_rulename". The "CAN_FIRE_rulename" occurs when the rule predicate (both implicit and explicit) conditions are all met. The "WILL_FIRE_rulename" occurs only when the scheduler allows the rule to fire [54]. Other powerful features include simulating until clock, single step, examining of signals and so on. Type-checking and schedule analysis are also available from Bluesim. Bluespec does not constrain user debugging and simulation to Bluesim. In fact, Bluespec is also able to support 3rd-party EDA simulators and waveform viewers such as nWave, gtkWave and ModelSim [43] through generation of a Value Change Dump (vcd) file. When the case-study was performed, Bluespec does not support any feature to enable automatic conversion of vcd files to Wave Log Format (wlf) for import into 3rd-party EDA simulators and waveform viewers. To automate this process, the author created a batch file and integrated the batch file for execution as a post-compiler command so as to allow non-circuit



Designs	Bluesim (secs)	Fastest RTL Simulator (secs)	Speedup (factor)
Wide GCD	5.53	39.7	7.18x
Life Game	8.37	283	33.9x
FIR Filter	4.23	61.8	14.6x
Upsize Converter	53.98	98.7	1.83x
Wallace Multiplier	48.66	417.0	8.57x
FIFO	11.73	124.2	10.6x
Mesa	37.45	126.4	3.37x
DIV3	8.24	116.8	14.2x
DES Core	28.55	89.36	3.13x
IDCT	15.69	16.32	1.04x

Figure 2.8: Bluespec Simulation Speed Comparison [6]

designers to verify and display the generated BSV design waveform in ModelSim. Although the absence of the file conversion feature is minor, it could affect the mass adoption of Bluespec for non-circuit designers.

In summary, BSV inherits its powerful abstraction features from advanced programming languages and offers rich user-defined polymorphic types and overloading, strong static type-checking, first class parameterization and higher-order programming, recursion and object-oriented (transactional) interfaces.

2.3.2 Xilinx System Generator

Xilinx System Generator (SysGen) for DSP is a system-level modeling tool which offers libraries, containing logic blocksets which are bit-true and cycle-accurate models, and is offered as a plug-in into MATLAB Simulink tool. With over 90 DSP building blocks are provided in the Xilinx DSP blockset for Simulink, these blocks leverage the Xilinx IP core generators to deliver optimized results for the target FPGA device. All of the downstream FPGA implementation steps, including synthesis and place and route, are automatically performed and an FPGA programming file is generated in the end for downloading into the FPGA platform.

Other features of SysGen includes System Resource Estimation to take full advantage of the FPGA resources, Hardware Co-Simulation and accelerated simulation through hardware–in–the–loop co-simulation, providing many orders of simulation performance increase. SysGen can also function as a system integration platform for the design of DSP FPGAs that allows the RTL, Simulink, MATLAB and C/C++ components of a DSP system to come together in a single simulation and implementation environment. As a result, error-free designs can be quickly prototyped in MATLAB environment and researchers only need focus on functional verification of their hardware architecture design.

In this thesis, the proposed SA hardware architecture is data-flow based and SysGen is the suitable graphical programming environment to help us achieve our research goals.

Chapter 3

Literature Review

In recent years, the increased proliferation of low-cost multi-processor systems has garnered growing interest from researchers wishing to utilize high performance computing platforms for high bandwidth control applications, such as ships, aerospace, robotics and automotive [44]. One popular approach is to offload high computational burdens onto reconfigurable hardware accelerator platforms such as Field Programmable Gate Arrays (FPGAs). When compared with custom application-specific integrated circuit (ASIC) designs, FPGAs are a preferred platform choice as they offer high design flexibility, shorter Time-To-Market (TTM) design cycles with low upfront non-recurring expenses (NRE). When compared with other popular high performance computing platforms, [45] has dispelled the common doubt of FPGA not being up to the task for handling real-time MPC.

In this chapter, the objective is to review the various approaches to realizing fast linear solvers for general scientific applications on FPGAs. At the same time, we will also review and present the various approaches to realizing fast linear solvers for MPC applications across various computing platforms. Finally, the findings are summarized and the proposed research contribution is briefly mentioned.

3.1 Linear Solver

3.1.1 Algorithm Mapping using High-Level tools for FPGA

The concept of matrix decomposition, using FPGA technology, for the purpose of hardware acceleration or application-specific applications is still an active research area with researchers [46–50] proposing the use of high-level methods to map the LU Decomposition algorithm onto FPGA architecture. Meanwhile the majority of researchers looked into application-specific architectures to accelerate the LU Decomposition techniques on FPGA. For example, [46] proposed the FPGA Regular Array Description language (FRADL) for automatic mapping while [47] demonstrated the use of Fortran to construct non-pivoted LU Decomposition. [48, 49] proposed design methodologies, such as bijective space-time transformation, to enable mapping of LU decomposition into linear systolic arrays with no reported implementations. [50, 51] proposed the use of their tool for automatic architecture generation and optimization. They used their proposed tool to implement a variety of matrix inversion solvers using Cholesky, QR, LU. They implemented their design on a Xilinx Virtex-4 XC4VSX35 FPGA for a 4 x 4 matrix using a 20-bits fixed-point precision and achieved a design operating frequency of 166MHz with a linear solver performance of approximately 0.35 million linear systems per second. In addition, [34] implemented their QR Decomposition-based Recursive Least Squares (QRD-RLS) design on a Xilinx Virtex-4 XC4VLX200 FPGA using floating point precision and reported a design operating frequency of 115MHz with a linear solver performance of 0.15 million linear systems per second.

3.1.2 LU Decomposition on FPGA

Mathematical pivoting is used in numerical algorithms, such as Gaussian elimination and LU Decomposition, to ensure numerical stability to the final result obtained. To do so, tracking of the sorted rows and columns in a matrix is required and moving of elements, within the matrix, adds performance overhead to the algorithm. As a result, the author noted that majority of the research efforts were focused on LU decomposition without pivoting until 2009. In this section, a brief literature review of LU Decomposition implementation variants, with and without pivoting, will be presented.

For implementations of LU Decomposition without pivoting, numerical stability for simpler design and better performance results is expected. A block LU decomposition concept was initially conceptualized and proposed by Prasanna [52]. Their design was implemented using 16-bit precision, on a Virtex 2 FPGA, and reported faster performance when benchmarked against similar implementations on a Texas Instruments (TI) DSP and Handel-C based Celoxica DK1. Prasanna's work was taken up by other researchers with proposed approaches for enhancing numerical accuracy, encouraging the use of high-level CAD and IP design tools for solving large matrix decomposition problems. For example, the largest LU matrix decomposition design size was 1024 x 1024 [53]. The design in [53] expands Prasanna's block LU decomposition numerical arithmetic to handle floating-point numerical calculations and is scalable for single large and multiple FPGAs. The result is a design which claims to outperform Prasanna's work and tuned software implementations including the ATLAS [54] and MKL [55] libraries on workstations by hiding the memory access latency behind the arithmetic computations. Similarly, [56] adapts Prasanna's block LU decomposition technique to achieve vector-multiplication blocks but no design synthesis results were reported. [57] and [58] proposed the extensive use of Xilinx's PlanAhead tool for efficient mapping of Prasanna's work onto FPGA. Design synthesis reports a sustained 8.5GFlops/s with 88.82% efficiency on XC5LX330T with reported operating frequency of 133MHz for solving a $16,384$ x $16,384$ matrix. [59] adapts Prasanna's work to solve linear systems of any size up to the capacity of off-chip system memory while numerical accuracy is handled by Altera's MegaCore IP function. Their proposed design claims to allow configuration of the number of processing elements in the design and benchmarked results shows their architecture can outperform a single processor by 2.2x and energy dissipated per computation factor is 5 times less. [60] proposed an implementation of LU decomposition using wavefront array with no implementation details.

[61] is the first to introduce a fine-grained pipelined LU decomposition with pivoting and exploits fine-grained pipeline parallelism to enable higher performance. A maximum of 19 PEs could be implemented on an Altera Stratix II EP2S130F1020C5 on a customized board and solved an 800 x 800 matrix in 149.13ms (6.71 linear systems per second). When compared to single Pentium 4 processor under the LINPACK [62] benchmark, a speedup of up to 6.14x and operating frequency of 96MHz were observed. On the other hand, three other independent research teams have proposed and demonstrated the use of unified processor architecture to perform various matrix decompositions while performance varies according to the number of PEs. In this context, the PE described is a hardware design component that performs custom data processing but does not exhibit the characteristics of a SA. [63] appear to be the first effort proposing an array to solve linear systems using LU decomposition with modified row-based pivoting strategy. The paper claims to be able to compute the inverse of an NxN square matrix in $(5N-2)$ steps and $2N$ steps when multiple matrix inverses are performed. They claim their architecture requires just $\left(Np+\left(\frac{N(N+1)}{2}\right)\right)$ PE arrays. Architecture feasibility was simulated in an Occam 2 system with no reports on hardware utilization. In [64], a linear array column-first tiled approach for LU and QR decomposition was proposed where data is loaded from off-chip memory, used for storing of matrix values. Their proposed design has an estimated sustained performance of approximately 9.8GFlops/s for LU decomposition with 200MHz operating frequency on XC5VLX330T FPGA. The same PE could be used to calculate QR decomposition with sustained performance of approximately 11GFlops/s for large matrices of up to 9000 elements exhibiting tall and thin characteristic. Similarly, [65] reported implementations of up to 4096 x 4096 and a maximum of 16 LU PEs can be synthesized on XC5VLX330T FPGA with 180MHz operating frequency using floating point.

3.1.3 Systolic Array implementations of LU Decomposition on FPGA

Gentleman and Kung first introduced the concept of Systolic Array (SA) to achieve data parallelism in [32] but it was clear that global communication would be an issue for large VLSI array designs. Mathematical pivoting is traditionally used to ensure an algorithm's numerical stability but Gentleman established that mathematical pivoting was more for storage minimization rather than global communication. This important observation led [32] to propose two triangularization concepts, triangularization with neighbor pivoting and orthogonal triangularization, with an aim of minimizing global communication. For the reader's reference, the concept of triangularization is to reduce matrix problems to solve for a system of linear equations and is only applicable for direct methods, such as LU Decomposition. Algorithms performing triangularization step incurs a computational complexity of $O\left(N^3\right)$ for general NxN matrix problem. The SA approach is known to reduce such computational complexity to $O\left(N\right)$.

Triangularization with neighbor pivoting was designed to avoid bottlenecking system performance due to global communication of pivot selection, required in Gaussian elimination algorithm. On the other hand, orthogonal triangularization was proposed for QR decomposition by Givens rotation. In the following years, different approaches using SAs for applications varying from simplifying matrix-vector/matrix-matrix arithmetic, surface fitting algorithms and matrix inversion architectures were proposed while others focused on exploiting the inherent parallelism in the LU decomposition algorithm.

The general research direction of some researchers were related to proposing an appropriate SA architecture to decompose a reasonably large matrix problem, with reasonable numerical accuracy, in the fastest time possible whilst requiring minimal FPGA hardware resources. For example, [66] proposed a hybrid systolic array which integrates a linear array with a 2D array. [67] introduced the concept of semi-systolic array and their largest design is 512 x 512 matrix with reported execution time of approximately 50ms (20 linear systems per second). [68] proposed a bi-dimensional linear systolic array which is non-modular and the PEs in their design are problem-size dependent. Their design performance is $2N(p+1)$ per unit time and has 50% efficiency. An LU decomposition SA with bounded broadcast was proposed by [69] and their architecture requires $\frac{N^2}{2}$ PEs. Each PE has 2 MPY-SUB units active on alternate cycles and claims speedup and efficiency of $\frac{kN^2}{(6k+3)}$ and $\frac{2k}{(6k+3)}$ respectively. Note that only boundary processors have special division operators. A systematic method for mapping LU decomposition of a matrix onto linear systolic arrays was proposed by [49] using the bijective Space-Time transformation method but no hardware synthesis results were presented.

3.2 Linear Solvers for MPC

[27] is the only work, to the author's knowledge, that performs a comparison study of linear solvers for MPC applications and they used the Gauss-Jordan elimination technique to solve for system of linear equations. [19] evaluated the computation performance, complexity and resource trade–offs between the use of the Interior Point Method (IPM) and Active Set Method (ASM) for FPGA implementations. [19] reports that IPM scales better for large problems while ASM is better suited for small problems and a clear definition of large and small problems is not explicitly mentioned.

The objective of this section is to provide readers an overview on various approaches to realize fast MPC Linear Solvers on embedded system platforms. Hence, a literature review is presented and the section is divided into FPGA-based Linear Solver and General Processor-based Linear Solver implementations for MPC applications.

3.2.1 General Processor-based Linear Solver for MPC

Some researchers [70–73] have presented implementations of MPC on PLCs able to solve 10 linear systems per second for MPC problems with prediction and control horizon of 10 and 3 respectively. Their work suggests that although PLCs may not be well-suited for fast dynamic processes that require sampling frequencies or linear solver speed of >10Hz, MPC can still be implemented in PLCs for static or slow processes.

[74–76] proposed the use of a Microcontroller, as the embedded computing platform, and their key assumption is low power consumption being a priority requirement for embedded MPC. No effort was mentioned or demonstrated on how to exploit the low-power modes on the respective Microcontroller to achieve a low–power embedded implementation of MPC. A 32–bit fixed-point implementation of Fast Gradient Method (FGM) targeting an ARM7 processor, with 48MHz clock frequency, was the fastest reported microcontroller implementation [76]. Their linear solver is able to solve 250 linear systems per second for a horizon length of $N = 20$ (or 20 x 20).

Recently, GPUs have been exploited for scientific processing applications through the use of high-level software tools, such as Compute Unified Device Architecture (CUDA) [77] and Open Computing Language (OpenCL) [78], with C-like programming syntax and compiler. GPUs boast a many core architecture with high memory bandwidth, making them well suited to address problems that can be expressed as data-parallel computation [79]. [79] appears to be the first work attempting to investigate the feasibility of employing GPUs as hardware accelerators for solving typical QP problems using Interior Point Method (IPM). A comparison of the data-parallel and problem-parallel approaches on GPU was also investigated for varying sizes of problems. The study revealed that the performance benefits of GPUs become apparent when several QP problems are solved in parallel on one GPU and each QP problem size has >100 decision variables. With regards to the GPU performance, the findings by [79] agree with [45]'s statement that theoretical peak performance of GPUs can only be achieved given sufficient parallelism in the application.

3.2.2 FPGA-based Linear solver Accelerators for MPC

FPGAs were notably the default platform choice due to their re-configurable nature, allowing researchers a platform to understand the resource and performance trade-offs for their proposed custom algorithms and hardware architectures. A survey on the effort in implementation of MPC on FPGAs has been conducted and is broadly classified into 4 major categories namely Algorithm Optimizations, Comparative Study of Linear Solvers for Parallel Architectures, Rapid Prototyping of MPC and Hardware Architectures.

A significant MPC algorithm optimization technique was introduced in [45] utilizing the very efficient and robust Conjugate Gradient Method (CGM) with deep–pipelining [80] to design a fast Quadratic Programming (QP) Solver. The implementation of IEEE single precision floating point CGM-based QP Solver reported a throughput of 35GFLOPS on a high performance XC5VLX330T FPGA, for matrix size $N = 58$ [81]. [82] pointed out the weakness of CGM and examines a more general iterative solver method using the MINRES algorithm. The circuit achieves 95% efficiency in practice with a sustained performance of 53GFLOPs on a XC5VLX330T FPGA and the reported performance is superior to any previous work. The already efficient MINRES performance was further improved by the pre–conditioned MINRES (PMINRES) method in [83].

[45] surveys the recent developments in parallel computer architectures to highlight the potential of such architectures for high-speed numerical computation, as required for on-line optimization of MPC. A comparative study of the popular Graphics Processor Units (GPUs), FPGAs and AMD Opteron 1220 Microprocessor [8] showed FPGAs being up to the task at handling MPC applications with fast dynamics. [45] has also pointed out that effort is still required to reformulate the

MPC problems further in order to obtain a trade-off between the amount of the parallelism an application requires and the ability to keep the deeply pipelined computational units operating at high efficiency. In addition, [45] have demonstrated and pointed out that there is no one numerical representation technique that can run efficiently on all MPC applications. To get the most out of an architecture, [45] encourages control theorists to work with digital electronics designers and computer architects to realize a new and exciting area of inter-disciplinary research.

[84] focused more on a development of a GUI Toolbox and rapid prototyping environment for deployment to various embedded hardware platforms. The authors of [84] claim their GUI toolbox features a modification of [28]'s QP Solver and report a 20% performance improvement over a carefully hand-coded implementation of an optimized QP Solver. In addition, [85] reports on their MPC implementation on various embedded hardware platforms ranging from microcontrollers, DSP chips and FPGAs, claiming that they were able to parallelize the MPC's arithmetic operation by using 3 floating point units (FPU) to perform matrix and vector operations on a Xilinx Spartan3 XC3S500E FPGA. Their iterative solver was coded in the C language and the FPGA-based solver performance is around 1000 linear systems for a problem size $N = 10$.

[86] adopted a more general approach to investigate hardware designs to take advantage of symmetrical and banded matrix structure, as well as methods to optimize the RAM use while increasing performance for larger order matrices [86]. A parameterizable circuit was developed for implementation of matrix-vector multiplications into existing hardware implementations of iterative methods. Results have demonstrated their proposed banded symmetric matrix using the MINRES solver with a traditional dot-product circuit provided the majority of the performance. The scalability and performance of the circuit could be improved through simple hardware changes to the circuit. While a traditional dot–product approach is unable to scale beyond a matrix order of 200, the proposed method was able to scale an order of slightly more than 500 matrix elements for implementations on a Xilinx

Virtex-5 LX330T FPGA.

Another approach is direct implementation of the MPC controller on FPGA. [44] focused on verifying the applicability of the "MPC on a chip″ idea in [87], [19] and [44]. [87] and [44] utilized the Handel–C high-level environment to synthesize a sequentially constrained MPC algorithm to a Celoxica RC200 and RC10 board respectively. Optimization, scheduling and parallelizing of the algorithm's operation, using IEEE single precision floating arithmetic, are left to Handel–C. [19] attempts to parallelize the previous work through ad–hoc programming in Handel–C and reported a 2x improvement in computational performance. [45] contributed similar effort. [88] compared the trade–offs between iteration count, computational precision, hardware utilization and execution time in accelerating the QP Solver for many small linear systems. Empirical results indicate that the 21–bit mantissa performs better than both double and single precision standard, with little to no penalty in the worst case. An average speed–up of 26x was reported for 10 x 10 problem.

The work in [89] utilized the ASM method and decoupled the already efficient Givens rotations for square root and division operations to only require multiplication and addition operations. Through careful selection of rotation points, Givens rotation operation can be computed in parallel. The end system design was described in VHDL and synthesized on an Altera Stratix EPSL150F115C2 FPGA with target frequency of 70MHz while occupying a total die area of approximately $5mm^2$. It is worth noting that a comparison of tuned bit-widths found that a 7–bit mantissa for solving QP problem and 15–bit mantissa for observer calculation yielded the best performance and precision tradeoff. Similar results were also observed in [90] with a focus on non-linear MPC and the tradeoffs between data word size and computation speed versus numerical precision and effectiveness of the computed control action. Unlike [89], [90] utilized the Householder Reflections method to exploit inner product calculation and outer product term of applying the Householder reflection method to the Jacobian Matrix. The simulated results of this approach found that acceptable results could be obtained with a mantissa of as low as 12–bits. [91] applies the experience learnt from [90] and [89] and applies the Conjugate Gradient Method (CGM) for solving the QP using the interior-point method for $\rho = \mathcal{H}^{-1}g$, the most computationally demanding step. A custom floating point format, 5 bit exponent and 15bit mantissa, demonstrated stable performance with the ability to solve QP problems in <30μs, despite the original 200μs limit. The design was described in VHDL and synthesized on an Altera Stratix EPSL150F115C2 FPGA with no mention of the design operating frequency.

The work in [92] and [93] utilizes the Conjugate Gradient method for matrix structure and the MINRES method to solve for linear systems. The papers propose a 2-stage hardware architecture, with the MINRES method as a dedicated parallel linear solver hardware resource in stage 2 of the architecture. In summary, the result was a 5x improvement in latency and a 40x improvement in throughout for large problems. [26] is an extension of the work in [92] and [93]. By entering the design in VHDL with deep pipelining, higher clock frequency was achieved over previous implementations and was based on IPM. The efficiency of the implementation resulted in performance figures that are several orders of magnitude larger than the ones considered in previous implementations. Similar work was also reported in [38] where a parameterizable FPGA architecture for linear MPC was described and their main source of acceleration was achieved through a parallel linear solver block. In addition, a new MPC formulation was presented to exploit the high throughput of their proposed FPGA architecture.

[94–96] employed the concept of coupling a co-processor to a microprocessor. [94, 95] hardware design involved a custom 16–bit Logarithmic Number System (LNS) for fast co-processor computation. The Newton's iteration method was adopted for their linear solver and hardware design was described in Verilog and targeted a Virtex–IV ML401 board. In addition, [94] reported their design size was 68% smaller than the design in [87] with a performance trade-off of 12%, which translates to lower power consumption. [96]'s work involves attaching a co–processor to a Xilinx 32–bit softcore microprocessor (Microblaze). The IPM method was developed in C language using IEEE single precision floating arithmetic and was synthesized onto a Virtex–II Pro FPGA. The result was a speed-up of 2.2x performance over a standalone implementation. [96] also revealed the speed-up was only achievable when the for-loop coding style had a tall and thin matrix A while the loop–unrolling style had a short and fat matrix A. Though all results suggest faster hardware computations, bus arbitration latency is a potential area for future research.

3.3 Summary

A summary of the previous work presented is shown in Table 3.1. From Table 3.1, highly optimized implementations of MPC linear solvers on FPGAs can be achieved using various approaches.

One such approach, which is typically reserved for experienced hardware design researchers, involves an embedded controller implementation using low-level language such as Hardware Descriptive Language (HDL). [26,92,98] proposed the adoption of primal dual interior-point method for fast computation of quadratic programming optimization problems. In order to exploit FPGA's massive parallelism, the authors resorted to fine-grained programming using HDL. In addition, [26, 45, 86, 92, 93, 98] investigated methods to take advantage of the structure of MPC problems, zoomed in to the issues at the memory access level and employs deep pipelining technique to achieve high performance. [90,94,99] resorted to manual mapping of their embedded controller design and much attention was paid to optimize physical design layout at the hardware level for the purpose of achieving application-specific hardware design that has fast performance, requires little hardware resource and low power consumption. On the other hand, [89, 91, 97] proposed the adoption of primal logarithmic-barrier interior-point method for solving the optimization problem. In this case, the authors paid special attention to the customization of floating-point number representation and various arithmetic operations to satisfy the control requirements for their flexible beam application. But their work is not as fine-grained as the

Table 3.1: Summary of previous work

	Related Work	Platform Device	Design Frequency (MHz)	Linear Solver Speed (10^6 lsps)	Reported Linear Solver Size	Programming Language	Numerical Precision	Numerical Method
General Linear Solver	[50, 51]	XC4VSX35T	253	1.8	4	Verilog	FXP(20,0)	LU
	[34]	XC4VLX200T	115	0.13	4	SysGen	FLP(6,14)	QRD-RLS
	[57, 58]	XC5VLX330T	133	8,500	16,384	Verilog	Double FLP	Block-LU
	[52]	XC2V1500T	120	NR	1,024	VHDL	NR	Block LU
	[53]	Virtex 5	250	7,850	1,024	Verilog	Single FLP	Block LU
	[59]	EP3SL340F1760C3	200	40,000	47,000	C + Verilog	Single FLP	LU
	[61]	EP2S130F1020C5	96	2,620	800	Verilog	Single FLP	LINPACK LU
	[64]	XC5VLX330T	200	9,800	NR	VHDL	Double FLP	Tiled LU
	[65]	XC5VLX330T	180	NR	4,096	Verilog	Single FLP	Fast Givens Rotation (FGR)
	[67]	XC2V500	129.53	NR	512	VHDL	NR	Strassen
MPC Linear Solver	[70–73]	Allen Bradley Rockwell PLC	NR	$10\text{x}10^{-6}$	10	C	NR	Gaussian Elimination (GE)
	[75]	STM32	24	$\sim 4.3\text{x}10^{-6}$	4	C	Single FLP	Gauss-Jordan
	[74, 76]	ARM7	48	$2.5\text{x}10^{-4}$	20	C	FXP(15,16)	Fast Gradient
	[79]	NVIDIA TESLA S1070	1,330	NR	20	C	Single FLP	Gauss-Jordan
	[81]	XC5VLX330T	287	35,000	58	Verilog	Single FLP	Conjugate Gradient (CG)
	[82]	XC5VLX330T	250	53,000	145	Verilog	Single FLP	MINRES
	[85]	XC3S500E	50	10^{-3}	10	C + HDL	Single FLP	Gauss-Jordan
	[19, 44]	XC2V3000	25	$8.1\text{x}10^{-3}$	10	Handel-C, MATLAB	Single FLP	Gauss Elimination
	[89, 91]	EPSL150F115C2	70	$3.3\text{x}10^{-2}$	12	VHDL	Custom FLP	CG + GR
	[97]	EPSL150F115C2	70	$7\text{x}10^{-2}$	16	VHDL	FLP(8,9)	CG
	[96]	XC2V1500	20			C + VHDL	Single FLP	GE
	[26, 92, 93]	XCVSX475T	150	0.71	15	VHDL	Single FLP	CG + MINRES
	[94, 95]	XC4VLX25	25	$1.13\text{x}10^{-3}$	10	Verilog	16-bit LNS	NR

former authors. At a coarse-grained level, [44, 96] reported attempts to parallelize the MPC algorithm through ad-hoc and systematic programming using high-level software tools, such as Handel-C and Xilinx EDK. Their works reported up to 2x computational performance improvement over previously reported implementations. Based on the information presented, the common observed disadvantage in this approach is design scalability and parameterization where a digital hardware design expert requires to be in the loop for any change in requirements, such as problem size and numerical word length.

Another approach is to consider the use of various numerical methods, such as Gaussian-Elimination, Givens rotation, QR Decomposition, Conjugate Gradient Method (CGM). For example, [86] examined the use MINRES algorithm as an iterative solver for cases where the A matrix is symmetric and not necessarily positive definite. CGM is employed while QR decomposition and Givens rotation were used to calculate the Lanczos vectors [86, 93]. In addition, the parallel linear solver in [92] is built upon the former's work but assumes A matrix is banded, symmetric but indefinite, with both positive and negative eigenvalues. [89, 91, 97] proposed CGM for use in their primal logarithmic-barrier interior point method to solve the optimization problem. Lastly, [96] employs Gaussian Elimination method and illustrated the performance difference readers can exploit for special matrix structures, tall-thin and short-fat A matrix. Based on the information presented, majority of the mentioned approaches exploit special properties of the A matrix in order to achieve fast MPC with deliberate attempts to avoid the use of division operation.

From the attempts described, a scalable hardware design using high-level tools to map LU decomposition SA architecture onto FPGAs remains an open research issue. To reduce the entry barrier and development time for implementing SA on FPGA, the high-level tool approach is deemed highly desirable by end-users. Although [50] developed their customized S&E tool for automatic architecture generation and optimization, end users are still expected to possess specialized HDL knowledge in order to design hardware with their tool. A benchmark of their high-level tool's generated output design with commercial software, such as Altera and Xilinx, are also absent. From literature, the use of SysGen for automatic architecture generation of parameterizable SA for LU decomposition is largely unexploited. In addition, no effort to implement mathematical pivoting more efficiently in hardware was observed.

In this thesis, the use of high-level software tools to overcome this expertise gap is proposed to assist non–circuit designers in rapidly prototyping a scalable and parameterizable linear solver which can be applied to other scientific applications. Secondly, the systolic array approach is proposed to enable efficient mapping of the linear solver onto FPGA hardware and computational complexity (for matrix decomposition step) is reduced from $\mathcal{O}(N^3)$ to $\mathcal{O}(N)$ due to parallelism. In addition, structural regularity of the systolic array architecture will provide researchers the ability to rapidly prototype a linear solver for varying sizes of N. Thirdly, a parameterizable triangular SA architecture approach to design, implement and validate using SysGen is proposed. Hence, non–circuit designers only need work at the architecture level with a design that has user configurable numerical word length. More importantly, a hardware design expert is not required to be in the loop. Fourthly, an initial prototype architecture is designed to perform LU decomposition for 3 x 3 problem before design scaling up to larger problem sizes for proof-of-concept purposes using SysGen.

The key research contribution is the exploitation of architecture parallelism, idle sequential cycles and omission of redundant arithmetic operations to achieve a novel systolic array hardware design which requires up to 50% less hardware resources when compared to similar work [51, 63]. Unlike most of the work reported, our proposed design architecture is able to achieve fast linear solver speeds and data throughput without side stepping the computationally expensive division operations. For the reader's reference, approximately 7 thousand linear systems was the fastest reported linear solver speed achieved in MPC applications [97] given a problem size of $N = 16$ with dense matrix structure.

Chapter 4

Case Study: High-Level Software Tool Selection

One of the objectives for carrying out the proposed research work is to understand and examine the hardware design decision high–level software tools make, in particular Bluespec and Xilinx SysGen. In this chapter, the Bluespec design rules will be exploited to generate a scalable yet optimized hardware architecture design.

From Algorithm 1, step 3 of the QP algorithm is computationally intensive where performance is dependent on the speed of the linear solver, essentially performance of the matrix–vector multiplication operation. If the hardware architecture is fixed, performance of the optimization solver will be affected when the matrices, to be multiplied, are of different sizes. Hence, a scalable linear solver with predictable performance should be investigated. Most hardware design optimization is assumed to be handled by the high–level software tool compiler, Bluespec and Xilinx SysGen respectively. Thereafter, the generated HDL design can be synthesized onto FPGA using synthesis tools such as Xilinx Synthesis Tool (XST) or Altera Synthesis Tool (AST). The results and design experience gained from both experiments, especially Bluespec's polymorphic feature, enable us to make better–informed design choices when describing the scalable optimization solver in high-level software tools. The option to re–use some of the existing modules/components, developed from both experiments, will help to decrease the development time needed for future researchers to implement the scalable optimization solvers.

This experiment was setup to compare and contrast the resource and performance trade-off between the Bluespec and Xilinx Sysgen IP Block implementation. The various Finite Impulse Response (FIR) Xilinx IP cores used in the design are:

1. Xilinx N–tap FIR Compiler
2. Xilinx MAC FIR Filter
    (a) Xilinx N–tap MAC FIR Filter
    (b) Xilinx 2N–tap MAC FIR Filter

The FIR filter design consists of a low-pass filter with coefficient weight of 0.8. Corresponding FIR filter coefficients were generated using MATLAB up to 512th order. The filter coefficients are in single precision format and were normalized up to 8 decimal places. All datapath size and data formats were standardized to fixed-point precision of Q(16,16), 16 integer and 16 binary points. All design files were synthesized for Virtex-6 XC6VLX315T FPGA hardware with speed grade "–3" selected and the balanced synthesis strategy was adopted for all synthesis results.

In the experiments conducted, the generated FIR filter design is put through XST with the purpose of understanding the advantages and disadvantages of Bluespec over traditional methods of hardware design. Likewise, the experimental results from the high-level tool implementation will contrast the performance and resource trade-off. The results obtained help us gain insights in creating an appropriate methodology, enabling non–circuit designers to effectively map their control algorithms, MPC in this case, onto any FPGAs. The mapping feature will be manifested in the form of an assessment process to determine and recommend the appropriate QP solver, based on the given control problem's size and constraints. The design experience gained from the investigation of Bluespecs polymorphism feature can be applied to the design of polymorphic QP solver(s) that is scalable and boasts rapid implementation.

4.1 Direct-form FIR Filter - Bluespec

In this section, a polymorphic direct–form FIR filter hardware design will be implemented as an example to evaluate the ease of defining parallelism using both the manual definition and Bluespec′s static elaboration method for various orders of an FIR filter. The polymorphic feature is achieved using Bluespec′s static elaboration method which simplifies the task of scheduling and parallelizing an N–tap FIR filter design to be as simple as modifying the values of N when instantiating. Unlike Bluespec, the traditional HDL approach would instead use "genvar" statements to generate structures and is coupled with manual effort to pipeline and debug for functional correctness and to identify race conditions – a time-consuming process. For benchmarking purposes, the hardware resource utilization and performance timing of a Bluespec described polymorphic direct–form FIR filter being compared against a similar design using MATLAB and is reported in the results section.

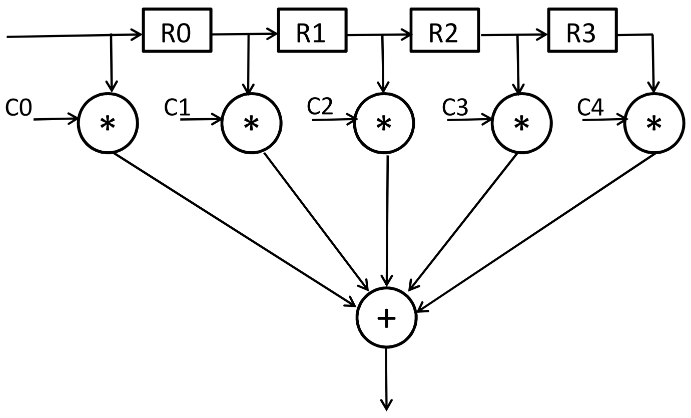


Figure 4.1: Example of a 4–tap FIR Filter

An FIR filter is a type of digital filter which operates on discrete-time signals and its simple implementation enables it to be commonly used in Digital Signal Processing (DSP) applications. The structure of a typical FIR filter is made up of simple component blocks such as multipliers, adders and delays, in the form of hardware registers, to create the filter's output. An illustration of a 4–tap FIR filter is shown in Figure 3.7. The frequency at which the FIR filter attenuates is determined by the FIR Filter's constant coefficients while the quality of the filter's response can be modified by increasing the number of taps, for a specific frequency response. It is straightforward to manually declare all the components for a small design system but the tasks become challenging when the size and complexity of the system design increases. When system performance constraints for a relatively large system are not achieved, techniques such as pipelining and deep pipelining are employed and further complicate the debugging process. This is where Bluespec′s powerful static elaboration allows researchers/engineers to generate a range of different hardware implementations from the same source code at no cost. Bluespec′s static elaboration method is required for IP core creation, for use in the later experiments.

To illustrate the parameterization capabilities of Bluespec, the amount of effort required for each method, manual declaration and static elaboration, will be quantified by the amount of time taken to modify the code in order to achieve the desired design size. A brief walk through of the respective coding methods will be performed for understanding. Lastly, the synthesis results for each method will be analyzed and discussed in the subsequent sub-sections.

4.1.1 Bluespec Manual Declaration

Figure 4.2 illustrates a code excerpt of a combinational-pipelined 8–tap FIR Filter design, module name is "mkFIRFilter″, described using BSV. Inclusion of external libraries and declaration files, such as filter coefficients, are easily achieved using Bluespecs import function. The instantiation of the external modules into the mkFIRFilter module is the same as type assignment descriptions when writing traditional HDLs. The logical bulk of mkFIRFilter contains a description of FIFO buffers, registers, accumulator register and other action assignment declarations.

The modification effort required can be quantified as (3N + 1) for size N of the FIR Filter – Manual declaration of N number of registers, N number of register assignments and (N+1) register assignments in the accumulator. The manual approach is prone to human–error and is only suitable for small problems of N in systems with low or no complexity. The built–in BlueSim simulator was also instrumental during the debugging process in identifying human errors such as incorrect register assignments.

```
module mkFIRFilter (AudioProcessor);

    FIFO#(Sample) infifo <- mkFIFO();
    FIFO#(Sample) outfifo <- mkFIFO();

    // 8 Registers will be Instantiated to implement the 2 mkFIFO modules Part 4.1-4.2
    Reg#(Sample) r0 <- mkReg(0);
    Reg#(Sample) r1 <- mkReg(0);
    Reg#(Sample) r2 <- mkReg(0);
    Reg#(Sample) r3 <- mkReg(0);
    Reg#(Sample) r4 <- mkReg(0);
    Reg#(Sample) r5 <- mkReg(0);
    Reg#(Sample) r6 <- mkReg(0);
    Reg#(Sample) r7 <- mkReg(0);

    rule process (True);
    Sample sample = infifo.first();

    // Simplest way to declare N registers to use
    r0 <= sample;
    r1 <= r0;
    r2 <= r1;
    r3 <= r2;
    r4 <= r3;
    r5 <= r4;
    r6 <= r5;
    r7 <= r6;

    // fromInt convert data from type Sample to type FixedPoint#(16,16) Part 4.1-4.2
    FixedPoint#(16,16) accumulate =
          c[0] * fromInt(sample)
        + c[1] * fromInt(r0)
        + c[2] * fromInt(r1)
        + c[3] * fromInt(r2)
        + c[4] * fromInt(r3)
        + c[5] * fromInt(r4)
        + c[6] * fromInt(r5)
        + c[7] * fromInt(r6)
        + c[8] * fromInt(r7);

    // fxptGetInt convert FixedPoint#(16,16) back to Sample  Part 4.1-4.2
    Sample sample_out = fxptGetInt(accumulate);

    outfifo.enq(sample_out); // Place sample on outgoing FIFO Part 4.1-4.2
    infifo.deq; // Removes input sample from the input FIFO Part 4.1-4.2
```

Figure 4.2: BSV Code for 8-tap FIR Filter using Manual Declaration

Although the process of manual declaration of an N–tap FIR filter was fairly straightforward, the process of modifying and debugging existing code became cumbersome enough to restrict the manual implementation method to 64-taps. For example, filter effort required for 64-tap FIR design is $\sim$ 192 minutes.

From Figure 4.3, the inefficient manual method illustrates that the exponential increase in resource consumed leads to a linear decrease in hardware performance – For a 64–tap FIR filter design, the simulated performance was 22.1MHz. Despite the less than stellar performance for a 64th order FIR filter, it is interesting to note that the synthesis design required 187 DSP48E1 cores/slices, the highest value observed across all experiments. A similar design using Bluespecs static elaboration method required 106 DSP48E1 cores/slices. For increasing order of the filter, the DSP48E1 slices/cores almost increased linearly with the exception for the 32nd order FIR Filter that saw a decrease from 57 to 25.

4.1.2 Bluespec Static Elaboration

The same FIR Filter BSV module was modified to be polymorphic and the first step was to modify the mkFIRFilter module interface to accept argument inputs that defines the filter size and the filter coefficients automatically, akin to modifying C code type function to accept input arguments. Next, the corresponding values of multipliers and registers, to be instantiated, are modified to correspond according to the input argument variable, tnp1.

In this experiment, a pipelined multiplier is implemented to improve the overall system throughput. The static elaboration method utilizes loop unrolling method and is able to automatically derive the required numbers of pipelined multipliers from the for loop statement, as shown in Figure 4.3. In contrast with the manual method of declaring combinational multiplier, the intuitive Bluespec compiler automatically unrolls a vector of multipliers, for values of N, time–saving technique. The same method was applied to the instantiation of registers required to hold the filter coefficients and the accumulation operation results.

The nature of the pipelined multiplier necessitates the need to invoke Bluespec Action operations – put data into the input buffer, in 1 clock cycle, and retrieve the multiplier's answer from the output buffer in another clock cycle; The multiplier module is assumed to take 1 clock cycle for multiplication operation. Therefore, it

```
import AudioProcessorTypes::*;
import FilterCoefficients::*;   // Part of Lab1 MIT - library from Bluespec  Part 4.1-4.2
import Multiplier::*;   // Part of Lab1 MIT - library from Bluespec  Part 4.1-4.2

typedef 8 Data_len;
Integer data_len=valueof(Data_len);
typedef 9 Num_mpys;
Integer num_mpys=valueof(Num_mpys);

module mkFIRFilter(Vector#(tnp1, FixedPoint#(16,16)) coeffs, AudioProcessor ifc); // 21 Apr :
Soln for lab2 Problem 5

    Integer numtaps=valueof(TSub#(tnp1,1)); //21 Apr : TSub performs subtraction of two numeric
    types (i.e numtaps = 8)
    Integer numpys=valueof(tnp1);

    Vector#(tnp1, Multiplier) v_multiplier <- replicateM(mkMultiplier());
    Vector#(tnp1, Reg#(Sample)) r <- replicateM(mkReg(0));

    rule putdata (True);
        Sample sample = infifo.first(); // Declared outside the rule on 03apr2012
        infifo.deq; // Removes input sample from the input FIFO Part 4.1-4.2

        // Static elaboration method to declare N registers for BlueSpec to figure out
        for (Integer i = 0; i < numtaps; i = i + 1) begin
        if (i == 0)
           r[i] <= sample;
        else
               r[i+1] <= r[i];
        end

        for (Integer i = 0; i < num_mpys; i = i + 1)    //9 MPYs
        begin
            if (i == 0)
               v_multiplier[i].putOperands(coeffs[i], sample);  // 21 Apr : Change c[i] to
               coeffs[i] based on vectorsize tnp1
            else
               v_multiplier[i].putOperands(coeffs[i], r[i-1]);  // 21 Apr : Change c[i] to
               coeffs[i]
        end

     endrule

    rule getdata (True); //(step < 'h8000);
    Vector#(tnp1, FixedPoint#(16,16)) ans; // Declare a vector - works! 6Apr12
        for (Integer i = 0; i < num_mpys; i = i + 1)
        begin
           ans[i] <- v_multiplier[i].getResult();
```

Figure 4.3: BSV Code for 8-tap FIR Filter using Static Elaboration

is only appropriate to split the main processing rule into 2 smaller rules, namely "*put_data*" and "*get_data*". This highlights the advantage of Bluespec where the splitting of rules enable the researcher/designer to concentrate on the main functionalities of each rule and leave Bluespec to figure out and schedule the respective logical blocks without user intervention. A case in point is the effort required by the researcher/designer for this experiment. The designer only needs to specify the values of N and Bluespec will intuitively figure out the resultant hardware design. In the event of race conditions, Bluespec compiler will either try to resolve such conflicts or provide clues for user to zoom into the particular code to fix. The built–in BlueSim simulation tool is also available as a debug console to help users to verify the functional correctness.

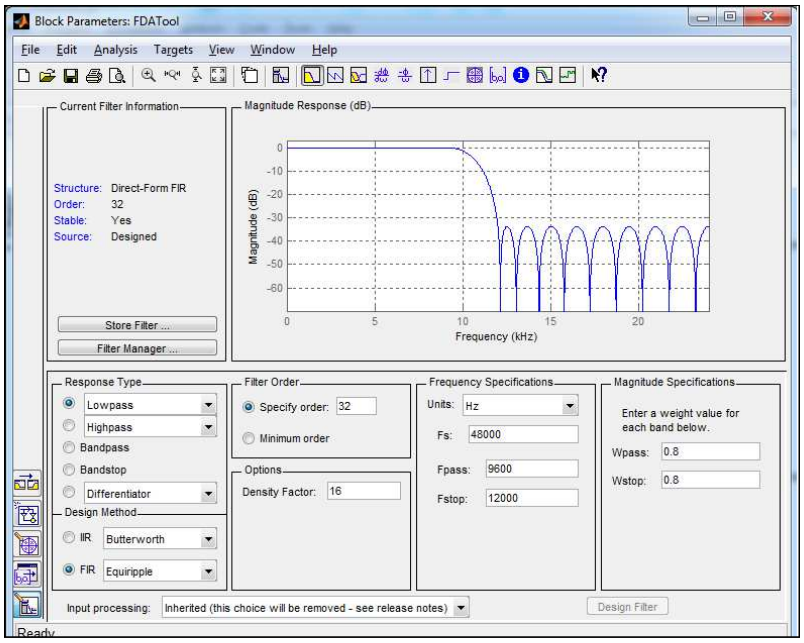
When this experiment was performed for values of N, the effort and time required to modify existing code was no more than the time needed to change for values of N. Such productivity enabled rapid implementation of FIR Filters up to 512–taps. It was noted that the time taken for Bluespec Compiler to elaborate the design and generate the corresponding Verilog output files was observed to take 15 minutes for 128th order design and about 1 hour for 256–tap design. During the design elaboration step for ≥128–taps, the compiler options had to be tweaked to override the default safety limit of 200k steps in order for the required Verilog design files to be generated. By extrapolating the previous steps required, a large buffer size of 2 billion steps had been set to synthesize the 512–tap FIR Filter design. The settings for the static elaboration steps will be investigated to resolve the need for such defining such large step buffers and reduce the time needed to synthesize the hardware design file as part of the future work.

4.2 Direct-form FIR Filter - MATLAB Xilinx SysGen

4.2.1 MATLAB FIR Compiler

Design of the FIR Filter was performed using MATLAB′s FDATool, which enables the digital filter designer to specify the response-type, filter order, frequency specifications and the weights, as shown in Figure 4.4. In this experiment, a low-pass direct-form FIR filter was selected with 0.8 as the weight coefficient, for corresponding N–taps. The filter′s output was verified using Simulink′s Spectrum Scope block upon software simulation.

To instantiate the design onto an FPGA, the Xilinx version of the Simulink FDATool must be included in the simulation and this is where elements of the software and hardware co–design come into effect. The FPGA boundaries are denoted by



Structure: Direct-Form FIR
Order: 32
Stable: Yes
Source: Designed

Response Type: [✓ Lowpass]

Design Method: [✓ FIR] Equiripple

Filter Order: [✓ Specify order:] 32

Options: Density Factor: 16

Frequency Specifications: Units: Hz; Fs: 48000; Fpass: 9600; Fstop: 12000

Magnitude Specifications: Enter a weight value for each band below. Wpass: 0.8; Wstop: 0.8

Input processing: Inherited (this choice will be removed - see release notes)

Figure 4.4: MATLAB Simulink FDATool Filter Design Tool

the Xilinx "Gateway In" and "Gateway Out" blocks while the FIR Filter design elements to be synthesized onto the FPGA are is illustrated in Figure 4.5. The task of the "Gateway In" block is to convert a floating-point input to a fixed–point format, in this case Q(16,16) format. The task for the "Gateway Out" block is to convert the FPGAs output back to double precision floating point. In this experiment, fixed-point Q(16,16) was chosen to enable a fair comparison with Bluespec's design implementation and the same FIR filter coefficient dataset, from Bluespec′s example of an 8-tap FIR, with a range of between -0.8 to 0.8. To generate filter coefficients for up to 512-tap, Simulink FDATool was used and the same data range constraints, from -0.8 to 0.8, was specified to avoid the numerical underflow and overflow issues for both BlueSpec and SysGen designs. An error rate comparison was performed in MATLAB on both the SyGen and Bluespec results and the largest error deviation was found to be 0.01643.

The "FIR Compiler" Simulink IP block was included to translate the FIR filter design, specified in the Xilinx FDATool, for synthesis onto any FPGA platform. The filter coefficients are generated and saved as a look–up–table onto the FPGA platform's on-board memory, reducing unnecessary time to compute the corresponding

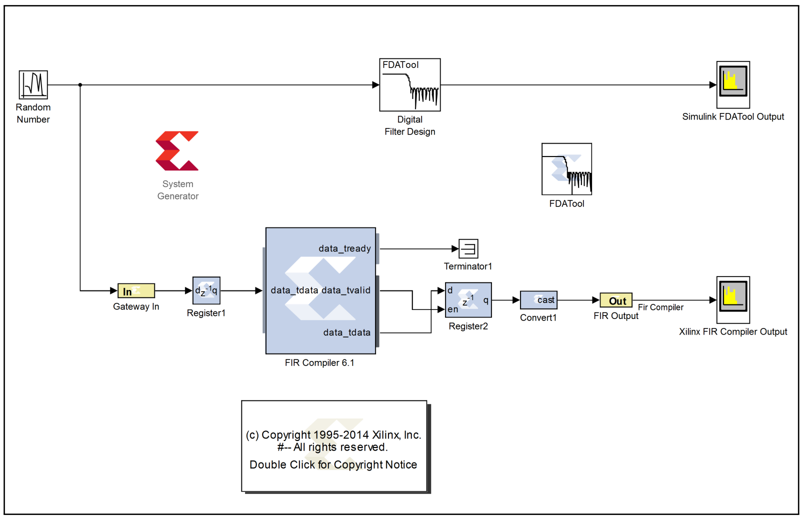


Figure 4.5: N-tap FIR Filter with Simulink FDATool Filter

filter coefficients online. For the purpose of a fair and accurate benchmark, the FIR Filter design has been pipelined at both the input and output of the "FIR Compiler" IP block. A capture register serves as a buffer to capture streaming data from the "FIR Compiler" IP block before new data can be typecast to an appropriate format, for display in Simulink Spectrum Scope tool block. To ensure a fair comparison with Bluespec generated design, all datapath format were verified to the Q(16,16) format. Finally, the System Generator token block was invoked to target the FIR Filter design onto a Xilinx XC6V315T FPGA hardware.

4.2.2 MATLAB MAC FIR Filter

Unlike the "FIR Compiler block", the designer only needs to modify Xilinx System Generators MAC FIR Filter IP block for a given design size, as shown in Figure 4.6. Then provide the coefficient values and specify the values of N to generate the desired MAC FIR Filter.

In this experiment, 2 MAC Filter designs were utilized, namely the N-tap and the $2N$-tap MAC FIR Filters. Regardless of the size of N or data throughput requirements, a compact hardware architecture, shown in Figure 4.7, is highly desirable for a MAC Filter design. The filter coefficient ROM obtains its data from a gen-

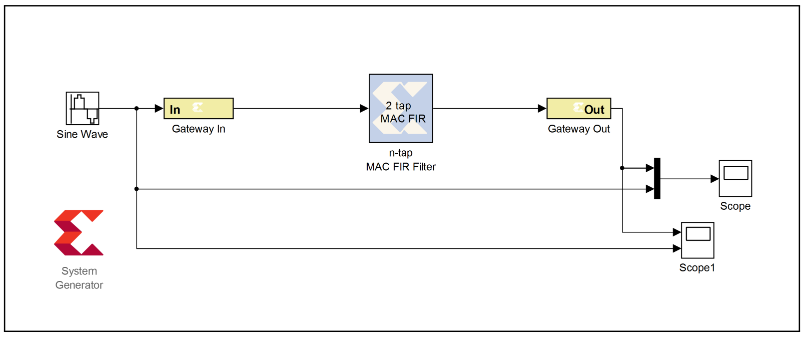
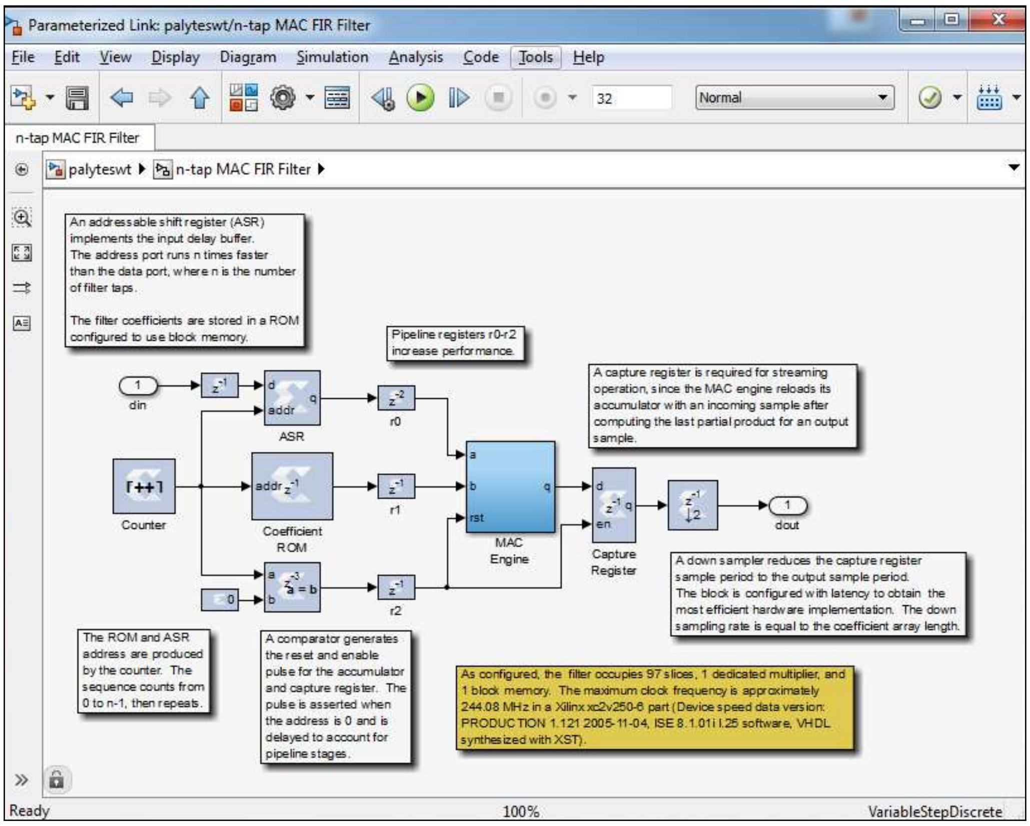


Figure 4.6: Simulink code for N-tap FIR Filter using Xilinx FIR Compiler 6.1 IP Block

erated data array containing data for memory, counter and down-sampling block parameters. Consequently, the model requires no modification to accommodate a change in the impulse response [29]. The hardware architecture of the $2N$-tap was a straightforward inclusion of an additional N-tap filter and a “Pre-adder” IP block in an attempt to improve the system’s overall throughput, see Figure 4.8.


Parameterized Link: palyteswt/n-tap MAC FIR Filter
File Edit View Display Diagram Simulation Analysis Code Tools Help
32
Normal
n-tap MAC FIR Filter
palyteswt ▸ n-tap MAC FIR Filter ▸
din
ASR
r0
Counter
Coefficient ROM
r1
r2
MAC Engine
Capture Register
dout
Ready
100%
VariableStepDiscrete

An addressable shift register (ASR) implements the input delay buffer. The address port runs n times faster than the data port, where n is the number of filter taps.

The filter coefficients are stored in a ROM configured to use block memory.

Pipeline registers r0-r2 increase performance.

A capture register is required for streaming operation, since the MAC engine reloads its accumulator with an incoming sample after computing the last partial product for an output sample.

A down sampler reduces the capture register sample period to the output sample period. The block is configured with latency to obtain the most efficient hardware implementation. The down sampling rate is equal to the coefficient array length.

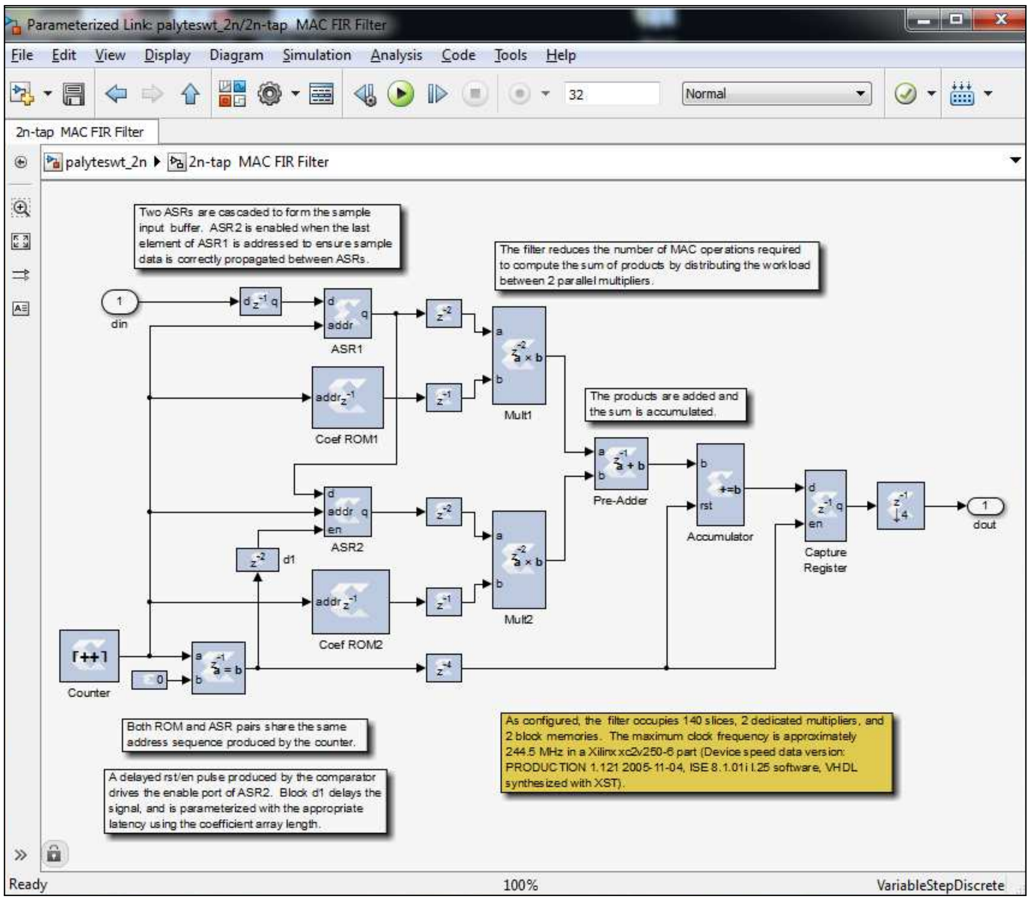
The ROM and ASR address are produced by the counter. The sequence counts from 0 to n-1, then repeats.

A comparator generates the reset and enable pulse for the accumulator and capture register. The pulse is asserted when the address is 0 and is delayed to account for pipeline stages.

As configured, the filter occupies 97 slices, 1 dedicated multiplier, and 1 block memory. The maximum clock frequency is approximately 244.08 MHz in a Xilinx xc2v250-6 part (Device speed data version: PRODUCTION 1.121 2005-11-04, ISE 8.1.01i I.25 software, VHDL synthesized with XST).

Figure 4.7: N-tap MAC Filter IP Block compact hardware architecture

The default FPGA boundaries are specified; input signal to the MAC FIR filter is



Two ASRs are cascaded to form the sample input buffer. ASR2 is enabled when the last element of ASR1 is addressed to ensure sample data is correctly propagated between ASRs.

The filter reduces the number of MAC operations required to compute the sum of products by distributing the workload between 2 parallel multipliers.

The products are added and the sum is accumulated.

Both ROM and ASR pairs share the same address sequence produced by the counter.

A delayed rst/en pulse produced by the comparator drives the enable port of ASR2. Block d1 delays the signal, and is parameterized with the appropriate latency using the coefficient array length.

As configured, the filter occupies 140 slices, 2 dedicated multipliers, and 2 block memories. The maximum clock frequency is approximately 244.5 MHz in a Xilinx xc2v250-6 part (Device speed data version: PRODUCTION 1.121 2005-11-04, ISE 8.1.01i I.25 software, VHDL synthesized with XST).

Figure 4.8: 2N-tap MAC Filter IP Block compact hardware architecture

from a sine wave generator while the output data is displayed using the scope block. The invoked "System Generator" token block will target the FIR Filter design onto a Xilinx XC6V315T FPGA hardware.

4.3 Results

4.3.1 Bluespec

From Figure 4.9a, the slice resource requirement has an exponential tendency. The slice resource trade–off was observed to be at least 2x more while the number of DSP48E1 cores was half of what is required for a similar order FIR design, using the manual elaboration method. Figure 4.9b also clearly illustrates a performance improvement between 2 to 4 times over the manual elaboration method for similar orders of FIR filter. Despite implementing pipelined FIR filter designs for both design approaches, the presented results clearly indicate the scalability and performance advantage of Bluespec′s static elaboration method. For example, the trade-off point for manual elaboration is at 16–taps and 128–taps for static elaboration approach. As a result, the static elaboration based approach enabled a 512–tap FIR filter design to be synthesized which required 546 DSP48E1 cores in order to achieve a design clock frequency of 58.92MHz. Lastly, intentional effort is required to declare design pipelining for the manual elaboration method while close to little or no effort is required for the static elaboration method.

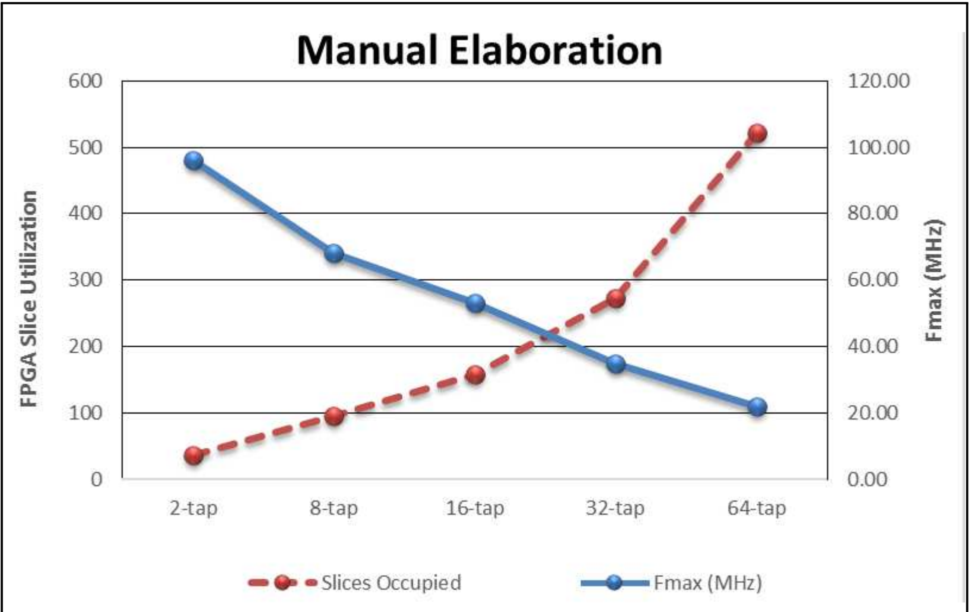
4.3.2 Xilinx FIR Filter Compiler

FPGA resource requirement has an exponential tendency with a 50% decrease in performance from 2–tap to 512–taps design. For example, a 512–tap FIR filter required 935 slices and 87 DSP48E1 slices in order to design clock frequency of 267.45MHz. When comparing the performance results for 512–tap FIR filter design, Xilinx's FIR Compiler method was at least 4x faster than both Bluespec implementations, see Figure 4.10. Similarly, the FPGA resources required was up to 86% lesser than a similar filter design described using Bluespec static elaboration method. Unlike Bluespec, the FIR Filter Compiler design is inherently pipelined and no user intervention is required.

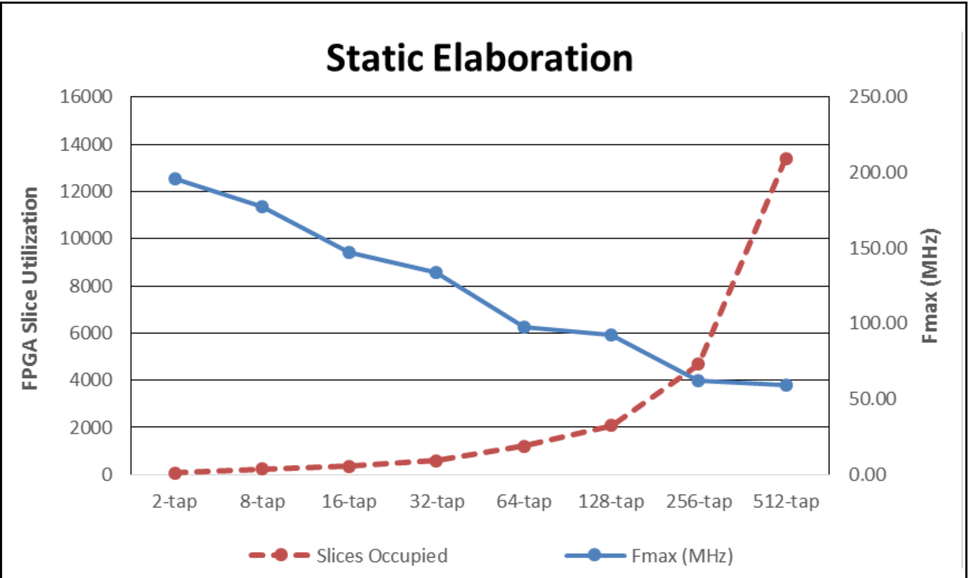
4.3.3 Xilinx MAC FIR Filter

Both MAC-based FIR Filter designs are inherently pipelined to exploit the built-in hardware resources of the DSP48 blocks.

For N–tap MAC–based FIR design, FPGA resource requirement was observed to decrease linearly while performance decreased linearly up to 512–tap. Overall, per-



(a) Manual Elaboration



(b) Static Elaboration

Figure 4.9: Bluespec Resource and Performance Trade–off for N–taps FIR Filter Design

formance decreased 46% from 2–tap to 512–taps design while the increase in resource requirements varied between 2–4x, see Figure 4.11. As expected, the resource requirements of the DSP48E1s remained constant across all N–tap FIR filter designs. The 2N–tap design implementation demonstrated similar resource and performance trends with the number of DSP48E1 remaining constantly at 2. Heavy exploitation

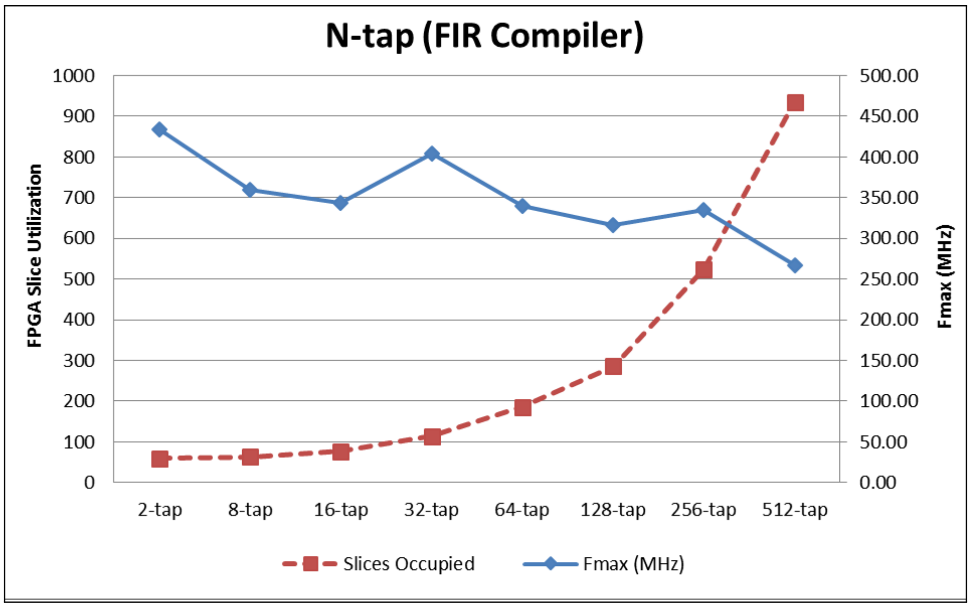


Figure 4.10: Resource and Performance Trade–off for N–taps FIR Filter Design using Xilinx FIR Compiler 6.1 IP Block

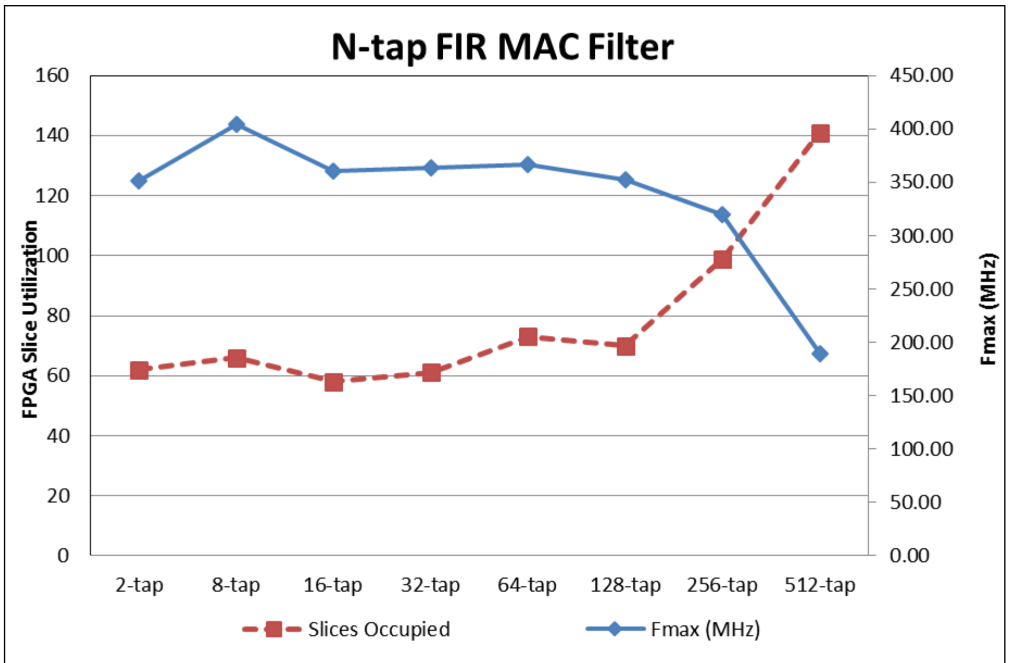
and dependence on DSP48E1 block limits the use of MAC FIR Filter IP blocks to FPGAs with embedded DSP48 blocks.

It was interesting to note that the 2N–tap design yielded an average performance improvement of approximately 11% and requires approximately 25% more FPGA hardware resources when compared against the N–tap design. The benefits of the extra DSP48E1 block in the 2N–tap design becomes advantageous for an FIR filter design ≥256–tap.

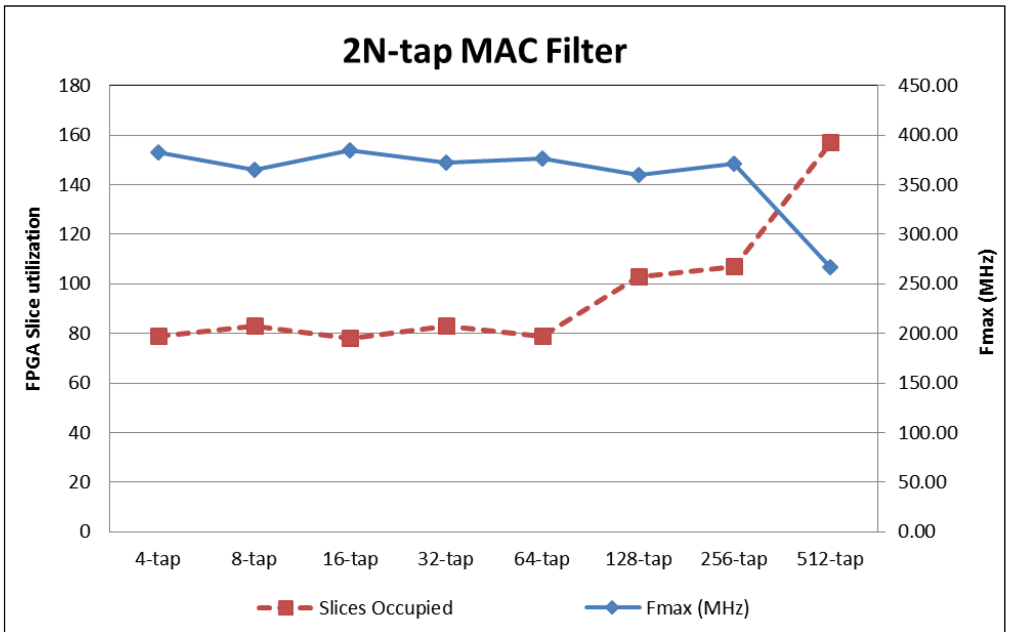
Lastly, we observed a sweet spot for the 2N–tap hardware design where it appears to be efficiently mapped onto the FPGA platform. For 64–tap and 256–tap FIR Filter designs, both designs incurred additional FPGA slice resource of approximately 8% with reported performance improvement of up to 16%.

4.4 Summary

The results suggest that Bluespec approach for mapping algorithms to hardware architecture is more flexible and FPGA platform neutral. Although the generated



(a) N–tap MAC Filter



(b) 2N–tap MAC Filter

Figure 4.11: Xilinx FIR Resource and Performance Trade–off for N–tap and 2N–tap MAC–based FIR Filter Design

hardware architecture is relatively scalable, both hardware resource utilization and performance scales in an unpredictable manner. From this case study, we had also found that the learning curve is much steeper than what is published and significant effort is required to achieve the good results reported. Although the Bluespec debugging environment supports 3rd party tools, such as ModelSim, a significant amount of setup effort and time was required. Moreover, debugging of signal lines required some amount of guesswork to verify the functional correctness of the FIR Filter hardware design.

Unlike Bluespec, the learning curve for SysGen was less steep and the provided hardware libraries enabled the FIR Filter design to be well-suited for the target FPGA platform. Design parameters, such as design bit-width, number of taps and frequency, of the FIR Filter design were labeled clearly and quick to configure. In addition, the Xilinx"s WaveformViewer block proved to be easy to use and useful as it enabled us to rapidly debug and validate the output results for the prototype FIR Filter designs. The MATLAB Simulink environment enable us to rapidly prototype the FIR Filter design graphically, prototype a scalable hardware architecture in a predictable manner and the corresponding data-flow of the hardware architecture design simplifies the debugging process. From this case study, we can conclude that SysGen is more suitable in helping us achieve our research objective – a suitable programming environment which enables domain experts or non-circuit designers to only work at the architecture level.

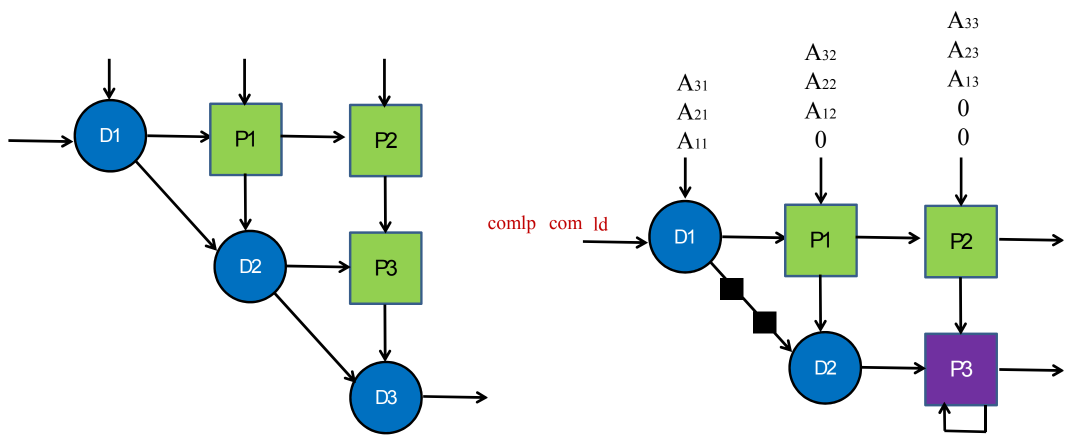
Chapter 5

Proposed Hardware Solver Architecture

In this chapter, we start by introducing the conventional structure of Triangular Systolic Array (TSA) hardware architecture and the basic Processing Elements (PEs). Secondly, we introduce our proposed TSA hardware architecture and use an example to illustrate how LU Decomposition can be performed using the proposed TSA. Thirdly, design details for our proposed TSA is described through the PEs. Fourthly, related work adopting a similar approach is briefly reviewed. Next, formulation of the latency estimation method for the proposed hardware design for both LU solver and linear solver is presented. Lastly, the key research contributions are highlighted.

5.1 TSA Hardware Building Blocks

Consider the TSA in Figure 5.1a to perform LU Decomposition on matrix A (see equation 5.1), where $N = 3$, and is hereby termed LU-TSA. Values of the A matrix are streamed into the TSA from the top, instruction sets are streamed in from the left and calculated values of L and U matrix are streamed out to the right of TSA, through the last PE (i.e. D3). Notice that the column-based values of the A matrix are fed into each column of PEs in a synchronous, delayed and orderly manner. TSA contains two types of PEs, *internal* and *boundary*. *Boundary* PEs (i.e D1 to D3) are designed to perform divide operations only and *internal* PEs (i.e P1 to P3) perform multiply-add-substract arithmetic operations. As this TSA design is universal to various matrix decomposition algorithms, PE-D3 may be required to perform divider operations while others requires simple multiply-add-subtract arithmetic operations.



D1
P1
P2
D2
P3
D3
A_{33} A_{23} A_{13} 0 0
A_{32} A_{22} A_{12} 0
A_{31} A_{21} A_{11}
comlp com ld
D1
P1
P2
D2
P3

(a) Existing **(b)** Proposed

Figure 5.1: Comparison of Triangular Systolic Array Architecture

TSA is usually proposed to reduce computational complexity of the matrix triangularization step, LU Decomposition in this thesis, to $\mathcal{O}(N)$. Conventional TSA hardware architecture consists of 2 PEs, namely Divider (Div) and Multiply-and-Subtract (Mult/Sub) PEs, see Figure 5.2. The PEs are typically named after the arithmetic operation the PE performs. Conventional TSA designs require $\frac{N(N+1)}{2}$ PEs and the values of L and U matrix are produced after $2N$ time-steps. Careful examination and mapping of computational operations for the LU Decomposition algorithm lead us to propose a TSA design which requires a total of $\left[\frac{N(N+1)}{2} - 1\right]$ PEs, see Figure 5.2.

From the implementation perspective, the hardware saving of 1 Divider PE is relatively significant as dividers are resource expensive to implement and require

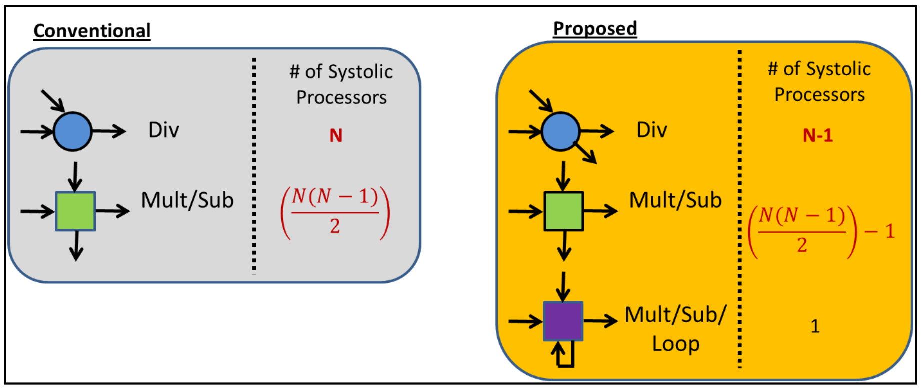


Figure 5.2: Comparison of Basic TSA Design and Components

much longer computational latency when compared to other PEs. The proposed divider hardware saving is trade-off at the expense of one additional time-step, needed for P3 PE to compute the correct value of U_{33} which would have otherwise been computed by D3, see Figure 5.1b. Existing TSA designs stream the values of L and U matrix out from D3, Figure 5.1a. But in our proposed design, respective values of L and U matrix are now streamed out from the last column of PEs, see Figure 5.1b. In the next section, we will use an example to illustrate the LU Decomposition operation on our proposed TSA design.

5.2 Example: LU-TSA Data Operation where N=3

In this example, data operations of our proposed LU-TSA design is illustrated in time-steps (t). Readers may assume a problem size of $N = 3$, where data is aligned beforehand and streamed into the respective PEs. For simplicity, the reader may assume all horizontal communication lines between PEs transport both instructions and data information while the vertical communication lines transport only data. Values of the A matrix are streamed into the SA in a column-based manner. For example, column 1 of matrix A is streamed into D1–PE; column 2 is streamed into P1–PE; column 3 is streamed into P2–PE.

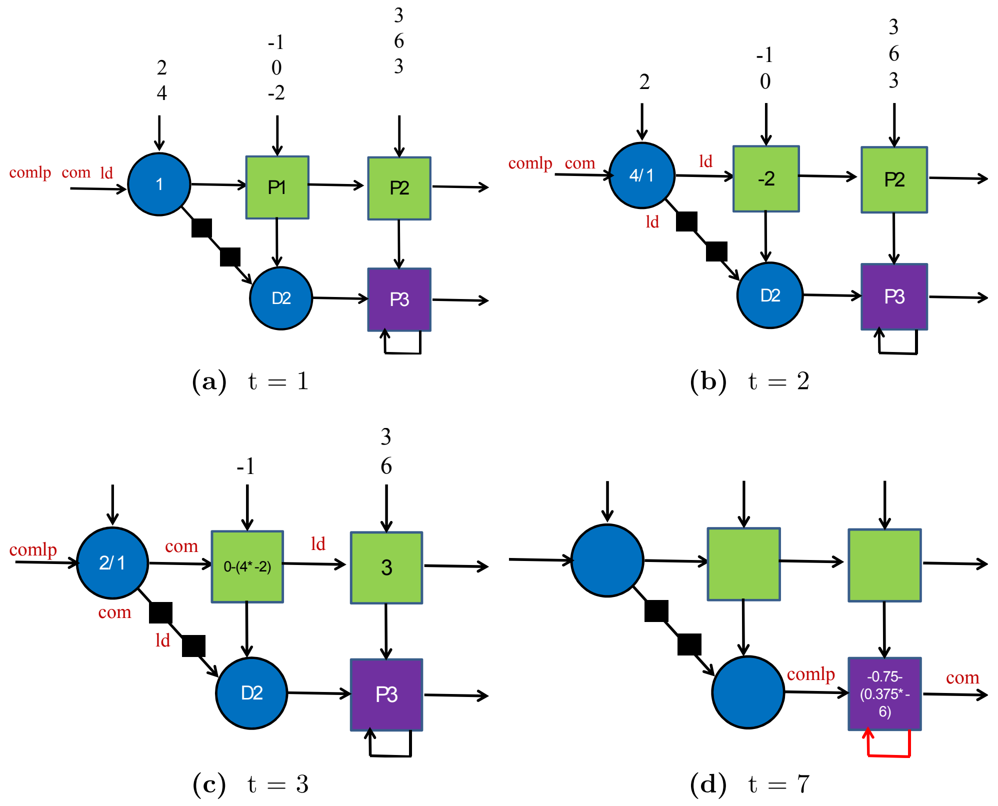


Figure 5.3: Example TSA DataFlow for 3 x 3 LU Decomposition at different time-steps

$$A = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ -2 \\ -1 \end{bmatrix} \tag{5.1}$$

At $t = 1$ (Figure 5.3a), instruction 'ld' informs D1 PE to perform a 'load' operation for data A_{11}, A_{11} is coefficient required for division operations at later time-steps. At $t = 2$ (Figure 5.3b), instruction 'com' informs D1 PE to perform a division operation on A_{21}(i.e $A_{21} \div A_{11}$) and value L_{21} is obtained. At the same time, 'ld' instruction is received by P1 PE and data A_{12} is stored. Readers may notice black boxes along the diagonal line connecting D1 and D2 PEs which denotes the presence of a unit delay element for instruction information. At $t = 3$ (Figure 5.3c), instruction 'comlp' informs D1 to perform a division operation (i.e $A_{31} \div A_{11}$) while instruction 'com' informs P1 to perform a multiply-subtract operation, calculating L_{31} and U_{22}

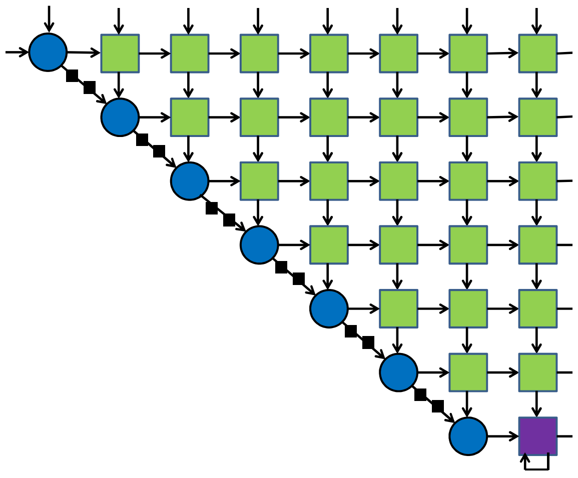


Figure 5.4: 8 x 8 TSA Design for LU Decomposition

respectively. The process is repeated in a synchronous manner across all the PEs in TSA to calculate all values of L and U matrix, except U_{33}, until $t = 7$. At $t = 7$ (Figure 5.3d), instruction 'comlp' informs P3 to re-use data values of L_{32} and U_{23} for multiplication and subtraction. Thereafter, the resultant value is subtracted from the intermediate value of U_{33} and the correct value of U_{33} is calculated as 1.5. Based on the command instructions, the calculated values of L and U matrix can be deterministically output from last column of PEs, see Table 5.1. On the other hand, total number of time-steps required to decompose a 3 x 3 problem is termed as the block latency and can be easily formulated as $2N + 1$, where N=matrix size.

Table 5.1: Order of calculated values of L and U matrix output for 3x3 TSA

		Command		
	Signal Line	1	2	3
Row1	D	L_{21}	L_{31}	
	X	U_{23}		U_{22}
Row2	D	L_{32}		
	X	U_{33}		

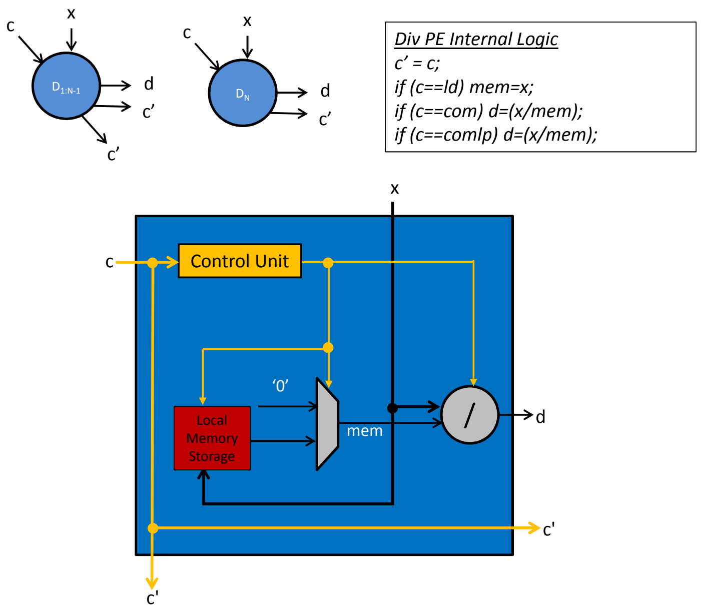
To illustrate the scalability and regularity of TSA design, we can consider a [8x8] design, see Figure 5.4. By utilizing the proposed PE Resource estimation $\left[\frac{N(N-1)}{2}\right]$ and TSA block latency formula $[2N + 1]$, 35 PEs (7 Div and 28 MultSubAdd) are required with a block latency of 17 time-steps, see Figure 5.4.

5.3 Processing Elements (PE) for LU-TSA

5.3.1 Divider PE

Existing work reviewed either implements custom hardware divider circuits or substitutes hardware division operations through the use Givens Rotation and CORDIC. To achieve our research objective, we propose the use of Divider IP cores for implementation on any FPGA platform.

The proposed TSA design requires $N-1$ Divider PEs for problem size N. Each Divider PE is designed with five inputs/outputs (I/Os); two inputs and three outputs with the exception of the $[N-1]^{th}$ Divider PE which has two outputs instead, see Figure 5.5. The divider PE consists of three major components: Control Unit, Local Memory Storage and Hardware Divider.



```
Div PE Internal Logic
c' = c;
if (c==ld) mem=x;
if (c==com) d=(x/mem);
if (c==comlp) d=(x/mem);
```

Figure 5.5: Divider PE

Instruction commands are streamed into and out of the Divider PE at each timestep. Similarly, the denominator and numerator values, required for the division operation, are streamed into the Divider PE via input x sequentially. The Control Unit operates as a state machine. Based on the incoming instruction command

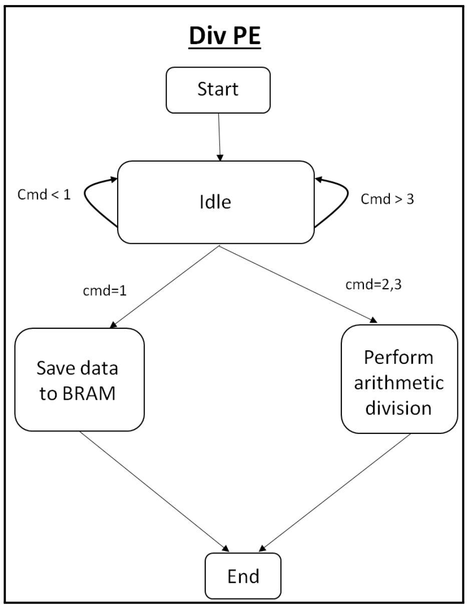


Figure 5.6: State Diagram for Div PE

received, the Control Unit outputs a series of commands to the other components within the Divider PE, such as toggling of multiplexer inputs or storing of values into local memory.

The state machine logic for the Divider PE is shown in Figure 5.6 and is illustrated using an example. Assume instruction *load* (cmd=1), is detected, a control signal will be transmitted to the local memory storage block to save the incoming value x, at memory address 1, for use at a different state. The same control signal informs the hardware divider to remain in idle or standby mode. On the other hand, instruction *com* (cmd=2) is detected, the Control Unit outputs a different control signal which informs the hardware divider to perform a division operation and utilize the incoming x value as the numerator. Similarly, the local memory storage receives a request to output the previously stored data as the denominator value for division operation by the hardware divider. The hardware divider performs the division operation and outputs a value, as output signal d, at the next time-step. Similarly, the received instruction command is output via signal c'. Lastly, instruction *comlp* (cmd=3) is detected and the Control Unit repeats the same operation where *com* instruction is detected. This is because instruction *comlp* has only specific effect on

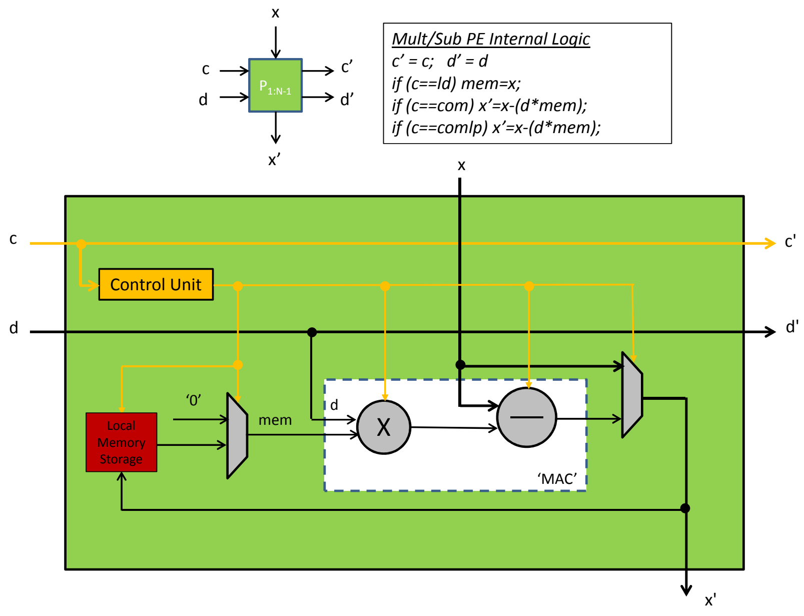


Figure 5.7: Multiply-Subtract (MS) PE

the modified multiply-subtract PE for the purpose of performing the loop function. At the same time, the local memory storage is pipelined and has a latency of 2 clock cycles.

5.3.2 Multiply-Subtract (MS) PE

The proposed TSA design requires $\left[\frac{N(N+1)}{2} - 1\right]$ MS PEs for problem size N. Each MS PE is designed with six inputs/outputs (I/Os), three inputs and outputs respectively, see Figure 5.7. The MS PE consists of three major components: Control Unit, Local Memory Storage and Hardware Multiply-Add-Subtract (MAC).

Similar to the Divider PE, instruction commands are streamed into and out of the MS PE from and out to neighboring PEs, at each time-step. Based on the incoming instruction command received, the Control Unit outputs a series of commands to the other components within the MS PE. For example, the *load* (cmd=1) instruction informs the local memory storage block to save the incoming value x, at a designated memory address, for use at a later time. On the other hand, if the *com* (cmd=2) instruction is received, the Control Unit informs the MAC unit

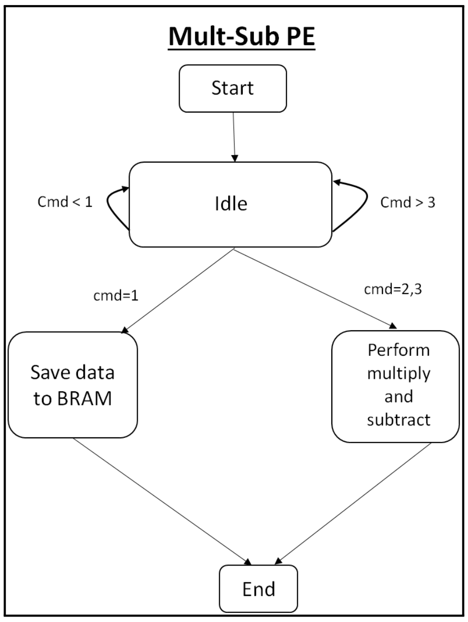


Figure 5.8: State Diagram for Multiply-Subtract PE

to perform a multiply-and-subtract operation while utilizing the values of *x,d,mem*. The calculated value is output via signal line *x'* for transmission to the adjacent PE block. The values of *c* and *d* are pipelined and concurrently forwarded from the input signal to the output signal lines *c'* and *d'*. Lastly, instruction *comlp* (cmd=3) is detected and the Control Unit repeats the same operation where *com* instruction is detected. The state machine logic for the Control Unit within the MS PE is illustrated in Figure 5.8.

Previous work reports custom arithmetic hardware circuits for implementation. To achieve our research objective, exploitation of on-board DSP48 cores is proposed to perform multiply-add-subtract arithmetic operations. The DSP48 core is internally pipelined and depending on the arithmetic operation performed, latency for MS PE ranges between 7–9 time-steps. Hence, the number of DSP48 cores and MS PE enables end users or researchers to perform quick assessment if their intended hardware design will be able to fit on a targeted FPGA platform, for a given problem size.

5.3.3 Modified Multiply-Subtract (MMS) PE

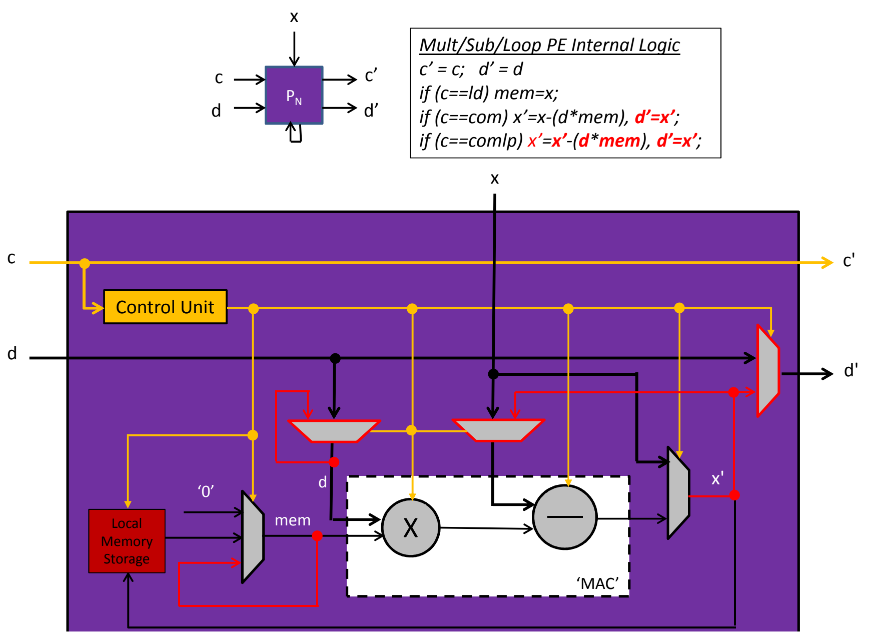

x
c
d
P_N
c'
d'
Mult/Sub/Loop PE Internal Logic
c' = c; d' = d
if (c==ld) mem=x;
if (c==com) x'=x-(d*mem), d'=x';
if (c==comlp) x'=x'-(d*mem), d'=x';
x
c
c'
Control Unit
d
d'
d
x'
'0'
mem
Local Memory Storage
X
—
'MAC'

Figure 5.9: Modified Multiply-Subtract (MMS) PE

The MMS PE can be considered the terminating block for the entire TSA design and the main components of the MMS PE are inherited from MS PE. The key difference lies in the inclusion of three additional multiplexers, increasing to a 3-input multiplexer for memory storage output signal line and the operational state logic within the Control Unit in order to perform a loop function, see Figure 5.9. For example, when instruction *com* (cmd=2) is detected, the MAC unit performs the multiply-subtract arithmetic operation. At the same time, the additional multiplexer in MMS PE is triggered to output values from x' input signal line. Meanwhile, additional multiplexers, placed at the input signal lines (d, x) to the MAC unit, will take the previous output value and route it back into the other signal input port with a delay of one time-step. Just to be clear, one input port for each of the three additional multiplexers are pipelined to re-use data that was output from the previous time-step. In the next time-step, the *compl* (cmd=3) instruction is detected and the Control Unit of MMS PE interprets that a loopback operation should be performed. Hence, the input multiplexers now output the previously output values of d and x into the input signal lines d, x of the MAC unit to perform the multiply-

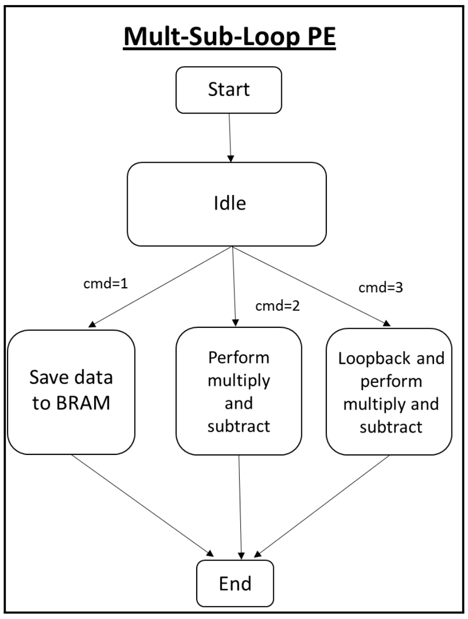
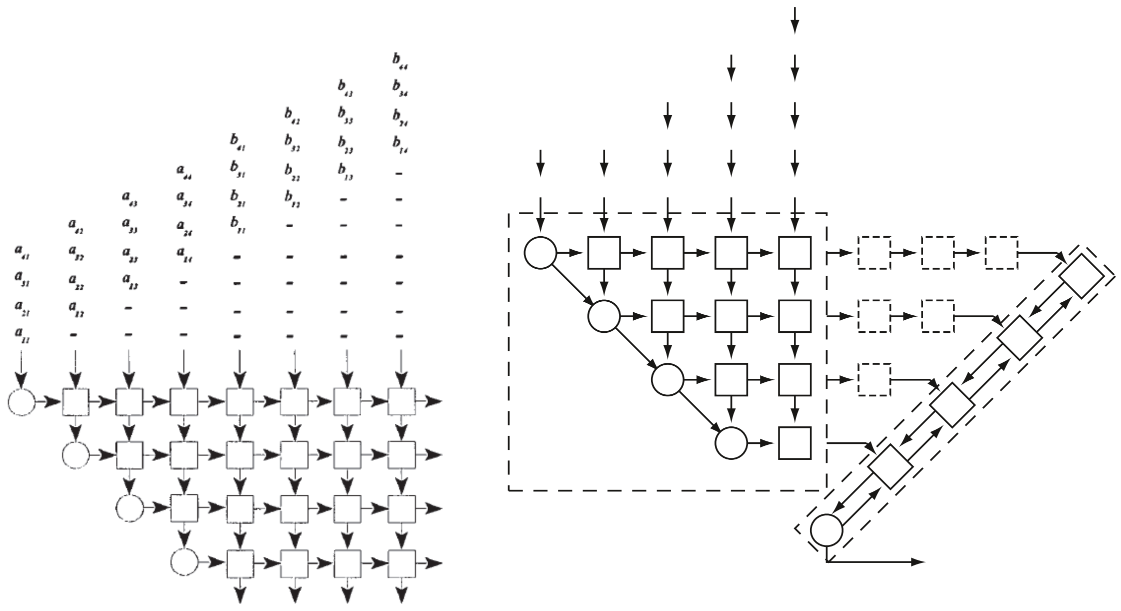

Mult-Sub-Loop PE
Start
Idle
cmd=1
cmd=2
cmd=3
Save data to BRAM
Perform multiply and subtract
Loopback and perform multiply and subtract
End

Figure 5.10: State Diagram for Modified-Multiply-Subtract PE

subtract arithmetic operation. The MMS PE Control Unit's operational logic is further illustrated in the form of state logic diagram, see Figure 5.10.

5.4 Triangular Systolic Array (TSA) Linear Solver

In addition to performing matrix decomposition using the LU method, both forward and back substitution steps are required to solve for a system of linear equations. The theoretical computational time required for the substitution steps is $2N^2$ time–steps and mainly consists of multiply-subtract and division operations. In [100], a linear solver design to solve a system of linear equations was discussed. Their design consists of a similar TSA design for orthogonal triangulation of a matrix using QR factorization while a triangular linear system is used to perform back substitution, see Figure 5.11b. Similarly, [63] proposed a linear solver for LU decomposition and forward substitution. Their design utilizes conventional TSA design for orthogonal triangulation of a matrix while a 2-dimensional (2D) rectangular SA is utilized for forward substitution, see Figure 5.11a.



(a) Wan's Solver [63] **(b)** Hu's Solver [100]

Figure 5.11: Existing Linear Solver designs using Systolic Arrays

To solve for a system of linear equations, [63] proposed that the back substitution step can be performed by reusing the TSA and 2D SA. The main advantage of using [63]'s design is that their design takes $6N + 2$ time-steps to compute the LU decomposition and forward substitution but the main disadvantage is that total latency is almost twice as long, $12N + 4$ time-steps, due to back substitution step. On the other hand, [100] did not present hardware nor performance values for their design. But the structural regularity of SA enabled the authors to estimate design latency for [100] and the results are presented in Table 5.2. Moreover, both SA-based linear solver designs by [63, 100] may not be possible to implement on a resource-constrained FPGA platform. For example, we can deduce from Table 5.2 that [63] design is highly unlikely to scale to $N = 16$ as the design requires additional N^2 PEs. [34] reports an FPGA implementation based on the design mentioned in [100] and was only able to scale up to $N = 10$ with estimates of up to $N = 12$. For any basis of comparison with similar work in the MPC community, the linear solver has to be at least $N = 16$ and is the main reason for choosing [97] for benchmarking purposes. In this thesis, the proposed linear solver design draws inspiration from both [63,100] work and propose techniques to overcomes limitations of the mentioned designs by exploiting idle sequential steps and removing redundant arithmetic operations to both the forward and substitution steps.

The reader is reminded that to solve for a system of linear equations of size N, LU decomposition, forward substitution and back substitution steps are performed. In the forward substitution step, N time-steps can be saved if one selects data value of '1' as the diagonal pivot value in the L matrix. This assumption is key to ensuring that time-consuming division operation does not require to be performed during this forward substitution step. Division and multiply-subtract arithmetic operations are performed for the back substitution step and is required to solve for values of $\hat{x} = U^{-1}\hat{y}$, where U is a matrix of $[N\text{x}N]$ dimensions and $\hat{y}$ is a an $[N\text{x}1]$vector. In order for division operations to occur in the back substitution step, values of $\hat{y}$ requires to be available. This suggests that backward substitution step is sequentially performed after forward substitution and no further parallelism is possible. But close observation of both substitution steps enabled the introduction of an ingeniously simple method to introduce parallelism in the substitution step, which was previously not possible or is strongly dependent on the software compiler's ability to parallelize the substitution steps.

5.5 Main Contributions

5.5.1 Exploitation of Serialized LU Decomposition

In this thesis, we propose that the diagonal division operations of the U matrix can be performed independent of the back substitution steps by means of the reciprocal operation. We are suggesting that reciprocal operation on diagonal U matrix be performed simultaneously during the forward substitution step. Hence, only the multiply and subtract operations should be performed during the actual back substitution step.

To illustrate, consider the values of matrix A defined in equation 5.1 which can be decomposed into L and U matrices respectively by LU decomposition method, equation 5.2. While the forward substitution step is performed to solve for L$\hat{y} = \hat{b}$, the reciprocal arithmetic operation can be concurrently performed on the diagonal values of the U matrix. Thereafter, the calculated reciprocal values are stored back into the U matrix and is termed U_{int}, equation 5.3.

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 2 & 0.375 & 1 \end{bmatrix} \quad U = \begin{bmatrix} 1 & -2 & 3 \\ 0 & 8 & -6 \\ 0 & 0 & 1.5 \end{bmatrix} \tag{5.2}$$

$$U_{int} = \begin{bmatrix} 1 & -2 & 3 \\ 0 & 0.125 & -6 \\ 0 & 0 & 0.67 \end{bmatrix} \tag{5.3}$$

$$U_{rot180} = \begin{bmatrix} 0.67 & 0 & 0 \\ -6 & 0.125 & 0 \\ 3 & -2 & 1 \end{bmatrix} \tag{5.4}$$

As a result of using the proposed method, the time-consuming division operation is partially replaced by the reciprocal operation whose latency can be hidden well-within forward substitution step and saves N time-steps, for problem sizes as small as $N = 4$. Lastly, the multiply-subtract arithmetic operations in the back substitution step is in fact a mirror image of the arithmetic operations in the forward substitution step. A 180 degree rotation of equation 5.3 enables the multiply-subtract operations to map perfectly onto each other, equation 5.4, and enables re-use of existing hardware resources.

The substitution step in both [63, 100] is implemented by either attaching a 1D SA or 2D SA, external to existing TSA design. The proposed hardware architecture requires a 1D SA to be integrated into existing TSA design architecture, allowing hardware design re-use at the expense of increased PE logic density and is only applicable for the last column of PEs. When compared to [63, 100] our proposed

Table 5.2: Latency and PE Comparison with Similar Work

	Proposed Linear Solver	Hu's Solver [100]	Wan's Solver [63]
Latency (time-steps)			
(non-pipelined)	**4N+2**	NR	12N+4
(pipelined)	$\boldsymbol{7N^2 + 56N - 38}$	NR	4M(3N-1)
# of PEs	$\boldsymbol{\frac{N(N+1)}{2} - 1}$	$\frac{N(N+1)}{2} + N$	$\frac{N(N+1)}{2} + N^2$

linear solver requires $\geq 2N$ fewer time-steps and $N^2 - 1$ fewer PE resources, see Table 5.2. Although the pipelined implementation is slower than similar work, it is a fair trade-off between performance and hardware resource utilization at the expense of requiring up to ∼50% fewer PE resources than similar work. As a result, larger linear solver problem sizes of $N>12$, which were previously not possible, can now be implemented on a resource-constrained FPGA platform. We wish to point out that no latency value (time-step) was reported by [100] and is denoted as NR in Table 5.2.

5.5.2 Data Throughput

TSA's natural ability to process new data at every time-step enables high data throughput. Consider the [3x3] example which implements a signed 18-bit fixed-point data precision. The theoretical LU solver performance can be formulated as $\frac{f_{max}}{N}$ LU problems/second, assuming new LU problem is introduced every N clock cycles, where N corresponds to the size of the matrix. Often, one unit time-step does not correspond to 1 clock cycle. This is because the actual time taken to perform division operations is usually much longer than simple arithmetic operations, so the block latency is multiple cycles.

In our design, the Divider block is pipelined with a design latency of 38 clock cycles while the Multiply-Subtract-Add block requires between 7 to 9 clock cycles, depending on the instruction set issued. Hence, the updated latency estimation is 45N and the sustained LU solver performance is reformulated as:

$$\boxed{\text{LU-TSA rate} = \frac{f_{max}}{45N} \text{ LU problems/second.}}$$

5.5.3 Speedup

SA Speedup is defined as the ratio of SA processing time (T) and single processor's processing time (T_s) of a given algorithm and can be formulated as $S = \frac{T_s}{T}$ [5]. Here, a larger speedup indicates the amount of parallelism inherent in the algorithm's design. A case in point is when we contrast our LU-TSA architecture with Gaussian Elimination algorithm implemented on a 32-bit Microblaze soft-core microprocessor running on a 100MHz Virtex II FPGA platform [96]. Design time-steps will be used as the basis of comparison to give readers an estimate on how well our hardware architecture design scales in terms of performance. In [96], their stand-alone Microblaze implementation reports $36N + M\,[122 + 64\,(N-1)]$ time-steps to compute and MxN problem, where M and N represents the number of rows and columns of the matrix respectively. In this thesis, we assume $M = N$ and their design latency can be further simplified to $64N^2 + 94N$ time-steps. Hence, our proposed architecture offers a speedup of one order of magnitude over [96].

5.5.4 Proposed TSA Linear Solver Architecture

Design architecture of the proposed TSA Linear Solver draws some inspiration from Gentleman's work, see Figure 5.11b. The proposed design is essentially an integration of a 1D SA into the last column of the TSA to overcome design limitations of Gentleman's work and is suitable for implementation of the proposed enhanced Linear Solver algorithm at the expense of a small increase in design complexity for the last column of PEs.

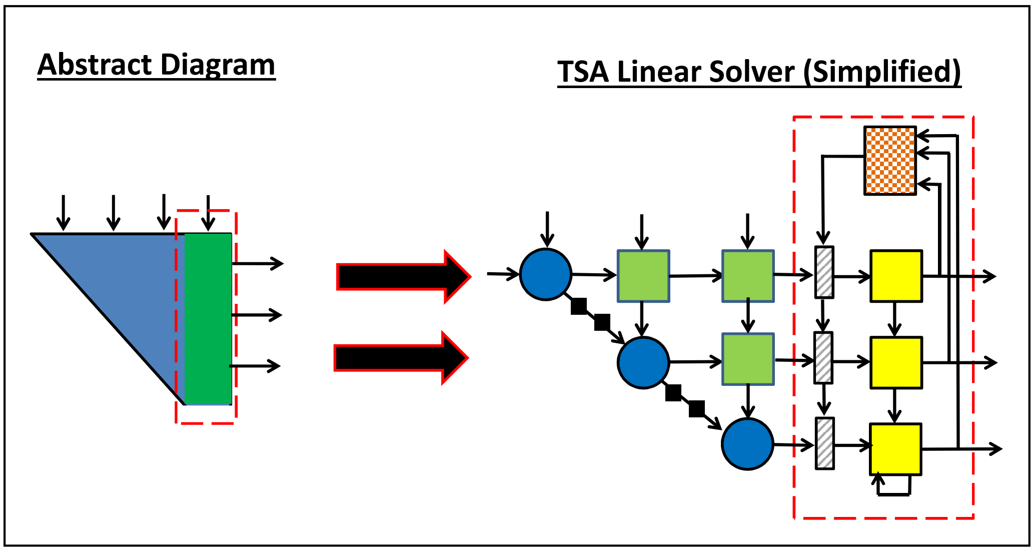


Figure 5.12: Simplified System Architecture for proposed TSA Linear Solver

The Data Router is denoted as a diagonally striped rectangular PE block in Figure 5.12. The block is responsible for routing data from its input data channels and corresponding data is selectively output on various output data channel lines. To side-step the introduction of slow combinational logic into existing TSA architecture, data and instruction set is now streamed out from the Data Arbiter PE Module, chequered PE block in Figure 5.12, into the input of the Data Router module of row 1, with both input and output pins pipelined. Assuming the data received needs to be forwarded to the Data Router PE in row 2, the incoming data will be delayed for a few time-steps so as to maintain data synchronization across all the PEs in the last column, readers are referred to signal L' in Figure 5.13. Hence, the characteristics of SA, such as design regularity, locality and rhythmic data communication, are not violated.

Within the Data Router PE, a small design consisting of multiplexers, comparators, Block RAM (BRAM) and delay elements are only required while the state machine controller logic manages the logical switching operations to ensure that the corresponding data is output on the correct data output line, see 'Control Unit' in Figure 5.13. As previously mentioned, the time-steps required practically is usually >1 and the BRAMs are utilized as temporary memory storage elements to hold

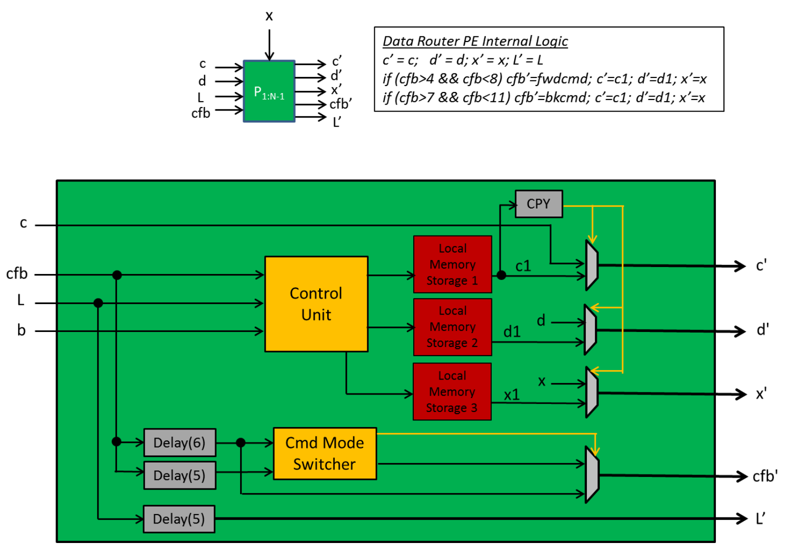


Figure 5.13: Data Router PE in proposed TSA Linear Solver

data elements for a finite time-step before the data is clocked out and new data is stored in the BRAM. In each Data Router PE, three memory elements or BRAMs are required to store all necessary data elements on each data output line and the regularity of this design enables the number of BRAM hardware resources to be quickly estimated for various problem sizes of N.

The Data Arbiter is denoted as a chequered PE block in Figure 5.12 and aligns its input data before forwarding to the aligned data to the MS PE, last column of the TSA architecture via the corresponding Data Router PEs, for the purpose of computing the L matrix, forward substitution step. In applying the proposed serialized algorithm exploitation method, the same process can be applied to compute the U Matrix, back substitution step, with only minor modifications to the state machine controller are required. For the MS PEs in the last column, the state machine controller's logic states requires to be upgraded to process and discern the operational states required for the forward and back substitution steps.

The lightly shaded PE in Figure 5.12 refers to the Multiply-Subtract Forward Substitution PE. This PE block is similar to the proposed Multiply-Subtract PE blocks with additional state logic, see Figure 5.14, to perform the forward and backward substitution operations, required to solve a system of linear equations using LU

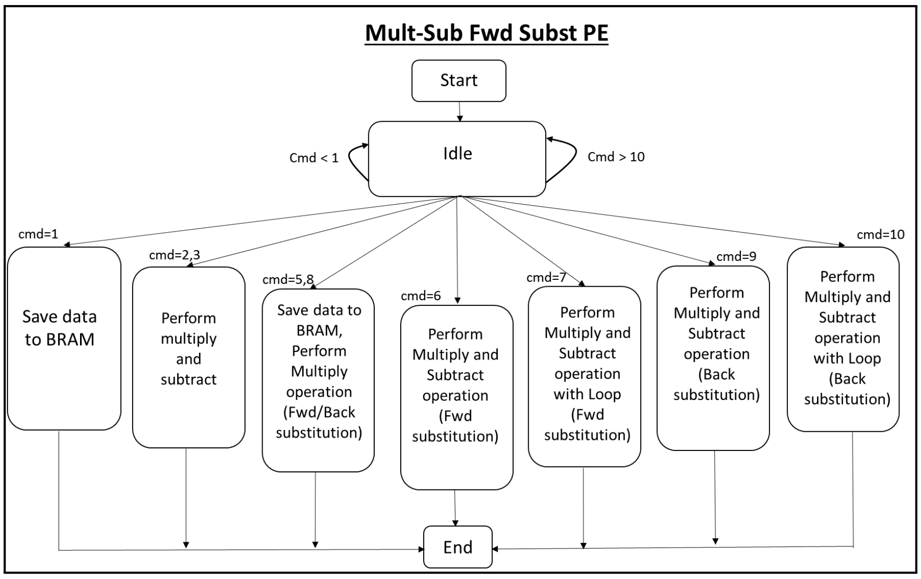

Mult-Sub Fwd Subst PE
Start
Idle
Cmd < 1
Cmd > 10
cmd=1
Save data to BRAM
cmd=2,3
Perform multiply and subtract
cmd=5,8
Save data to BRAM, Perform Multiply operation (Fwd/Back substitution)
cmd=6
Perform Multiply and Subtract operation (Fwd substitution)
cmd=7
Perform Multiply and Subtract operation with Loop (Fwd substitution)
cmd=9
Perform Multiply and Subtract operation (Back substitution)
cmd=10
Perform Multiply and Subtract operation with Loop (Back substitution)
End

Figure 5.14: State Diagram for Multiply-Subtract Forward Substitution PE in proposed TSA Linear Solver

Decomposition. Although we proposed the idea that backward substitution operations can be directly mapped onto forward substitution, we wish to point out that subtle arithmetic operation differences exist. Abstractly, these differences are handled by the introducing additional control states to the Control Unit, states 5–10, and upgrading the DSP48 data input multiplexers to 4-input from 3-input. States 5 to 7 are for forward substitution while states 8 to 10 are for backward substitution.

As a result of our innovation, larger linear solver problem sizes of N>12, which were previously not possible, can now be implemented on a resource-constrained FPGA platform and the pipelined linear solver performance can be formulated as:

$$\boxed{\text{Linear Solver performance} = \frac{f_{max}}{7N^2+56N-38} \text{ linear equations/second}}$$

Chapter 6

Implementation Setup and Results

In this chapter, hardware resource and performance results for the proposed TSA-based solvers will be presented and contrasted against similar work.

6.1 System Setup

The proposed LU-TSA, Figure 6.1, and linear solver, Figure 6.2, designs were implemented using SysGen [23] software environment targeting the mid-range Xilinx Virtex 6 FPGA (XC6VLX240T) for the purpose of design verification. Numerical accuracy is strongly dependent on the control application's requirements and we chose to adopt the numerical precision in [97] for benchmarking purposes. Use of a floating-point number format heavily impacts performance and consumes large amounts of hardware resources. Hence, our proposed design implements signed fixed-point number format with 9 integer bits and 8 fractional bits with numerical precision of approximately 2×10^{-3}, similar to that reported in [97]. Unlike [97], our design is word length and matrix size parametrisable at the PE level within the SA architecture.

Instead of developing a customized fixed-point divider, Xilinx's Floating-Point Divider IP core is used in conjunction with conversion blocks for switching between

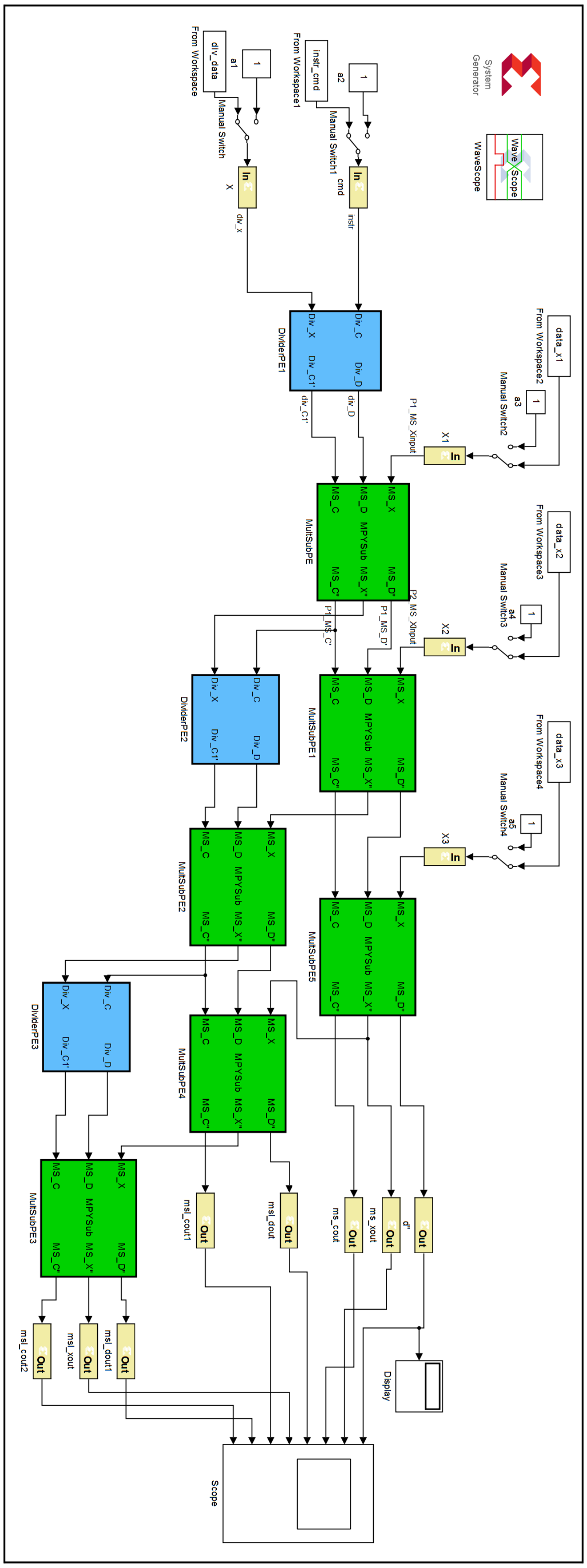


Figure 6.1: LU-TSA Hardware Architecture

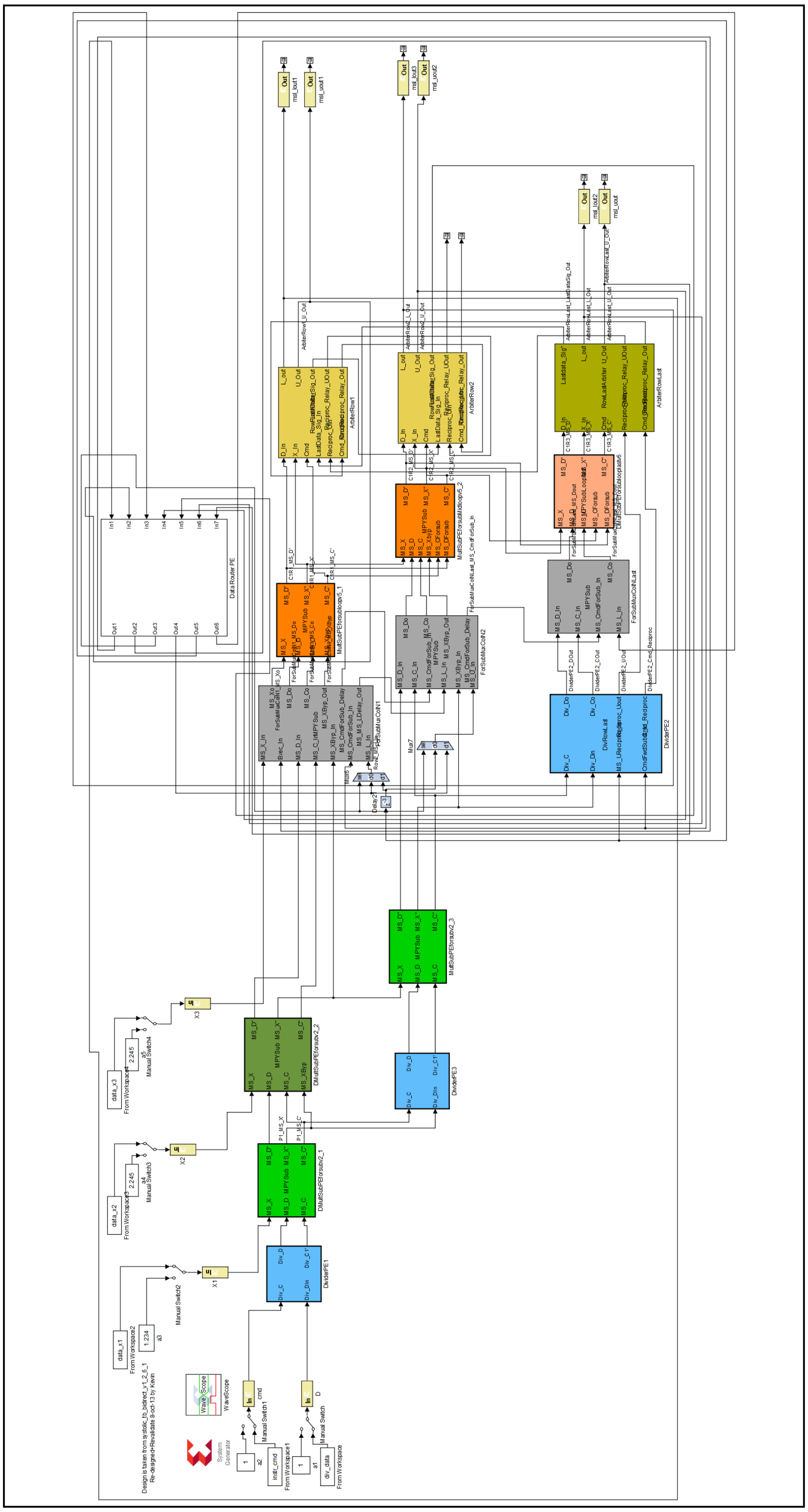


Figure 6.2: Proposed Linear Solver Architecture

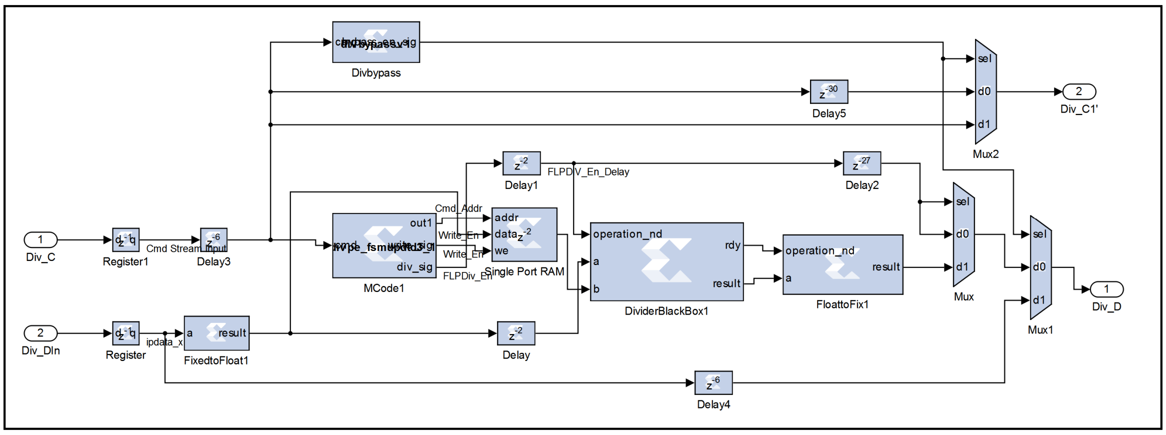


Figure 6.3: Internal logical blocks for Divider PE

fixed-point and floating-point. Tests revealed that the proposed divider is able to accept new data at every clock cycle and has a fixed data latency. Careful exploitation of this knowledge and idle sequential cycles, within the back substitution step, enables a saving of N time-steps. A single port RAM is configured to store the denominator data value, required for division operations after the numerator data becomes available. To manage the various operational states within the Divider PE, a state machine controller is programmed using MATLAB code and is incorporated as an M-code block in SysGen, see Figure 6.3.

In SysGen, three different DSP48 blocks may be used to instantiate the on-board DSP resources. The DSP48 Macro block is selected as it provides an abstract interface to DSP resources, enables ease of use, code readability and portability across various FPGA platforms. For the Multiply-Subtract PE block, the DSP48 macro block is also exploited to function as an instruction-based processor to perform user defined arithmetic operations and is set for automatic pipelining. The macro block allow designers to configure design latency and specify user defined arithmetic instructions for the DSP resource to execute. For example, assume the DSP48E1 is instantiated on the Virtex 6 FPGA platform. The DSP48E1s have 3 inputs (A,B,C) and 1 output each. At $t = 1$, instruction code '1' is issued and DSP48E1 performs $A \times B$ arithmetic operation. At $t = 2$, instruction code '2' is issued and DSP48E1 performs $C - (A \times B)$ arithmetic operation simultaneously. Given a

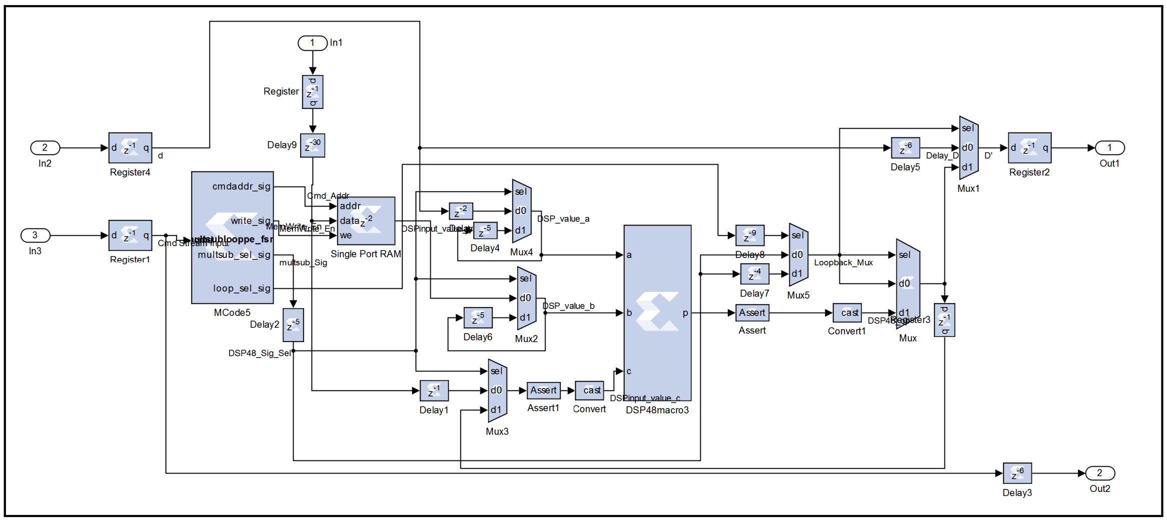


Figure 6.4: Internal logical blocks for Multiply-Subtract PE

pipelined implementation, a minimum latency of 4 clock cycles is required for the DSP48E1 to complete the required arithmetic operations. Similar to the Divider PE, a single port RAM is required and configured to store temporary variables for multiply and subtract operations at a later time–step to calculate the intermediate or final values of the L and U matrices. The key difference between the multiply–subtract, see Figure 6.4, and multiply–subtract–loop PE is the requirement for the latter to calculate the value of U_{NN}. To achieve the loop function, an additional multiplexer, *Mux3*, is connected to input C of the DSP48 macro block while the second input of *Mux3* is connected to the output of the DSP48 macro block with a delay of one time–step.

6.2 Results and Discussion

6.2.1 LU-TSA

From Table 6.1, the block latency formula has an average and maximum estimation error of approximately 1.3% and 1.9% respectively. Compared with [51], our LU-TSA solver latency is approximately 7% faster and solver throughput is approx-

Table 6.1: LU-TSA Performance & Resource Benchmarking

		LU TSA (4x4)	**LU TSA (16x16)**	Custom LU HW (4x4) [51]
Performance	Word length	**Hybrid(18,9)**	**Hybrid(18,9)**	FXP(20,0)
	Latency(clock cycles)	**134**	**675**	142
	Clock Frequency(MHz)	**~508**	**~476**	253
	LU Solver Speed (Mlsps)	**~3.8**	**~0.71**	1.8
Resource	Slices	**935**	**8,287**	709
	RAM18E1	**9**	**135**	1
	DSP48E1/DSP48	**6**	**120**	4
FPGA	Device Type	**XC6VLX240T**	**XC6VLX240T**	XC4VSX35T

imately 2.1x faster for a problem size of $N = 4$, see Table 6.1. A trade-off exists as LU-TSA consumes approximately 1.5x more hardware resources with an exception of BRAMs. From the results presented and structural regularity of LU-TSA solver, we can estimate that $N = 64$ LU-TSA design can be implemented on a Virtex 6 XC6VSX475T while $N = 128$ can be implemented on a Virtex 7 XC7V2000T.

6.2.2 Linear Solver

The proposed TSA-based linear solver requires $\frac{N(N+1)}{2} - 1$ PE resources. Compared to [51] in Table 6.2, our design is 2.3x faster and approximately 5% smaller than the custom linear solver reported in [51] with an exception of BRAM and DSP block usage.

Similarly, in comparison with similar work by [63], our proposed design requires 2N fewer clock cycles and $N^2 - 1$ fewer PE resources. Our design is one order of magnitude smaller in size and is estimated to provide resource savings of up to

Table 6.2: Linear Solver Performance & Hardware Resource Benchmarking

		Linear Solver (4x4)	**Linear Solver (16x16)**	Custom HW (4x4) [51]	MINRES (16x16) [101]
Performance	Word length	**Hybrid(18,9)**	**Hybrid(18,9)**	FXP(20,0)	Floating-Point
	Latency(clock cycles)	**304**	**2,650**	473	374
	Clock Frequency(MHz)	**∼247**	**∼198**	166	∼250
	Linear Solver Speed (Mlsps)	**∼0.82**	**∼0.08**	∼0.35	∼0.04 to 0.68
Resource	Slices	**1,933**	**15,622**	2,025	∼12,500
	BRAM/RAM18E1	**17**	**167**	1	∼37
	DSP48E1/DSP48	**6**	**120**	12	∼40
FPGA	Device Type	**XC6VLX240T**	**XC6VLX240T**	XC4VSX35T	XC5LX330T

50% for large sizes of N. In [63] and [51], $N = 10$ was the largest problem size that could be implemented on FPGA. By contrast, our linear solver can at least implement a design size of $N = 16$ on the smallest Virtex 6 (XC6VLX75T). Based on the results in Table 6.2 and structural regularity of our proposed design, we can confidently estimate our linear solver design can be implemented for up to $N = 32$ on XC6VSX475T and $N = 64$ on Virtex 7 XC7V2000T respectively. In retrospect, approximately 7 thousand linear systems per second was the fastest reported linear solver speed achieved in MPC applications in [97] given a problem size of $N = 16$. When compared with our SA-based linear solver, our approach is approximately 12x faster and we estimate that our hybrid arithmetic implementation, floating-point and fixed-point, requires lesser hardware resources than the floating-point implementation in [97].

Reported work in [98] is built upon [101]'s parallel linear solver. [101] proposed an iterative floating-point linear solver to solve a system of linear equations with dense data type and the maximum iteration count to reach a solution is N. On the other hand, our proposed linear solver utilizes both floating-point and fixed-point(18,9) for dense data type. For $N = 16$, our novel TSA architecture design is up to 2x faster for their worst case scenario, see Table 6.2. But our design requires approximately 25% more hardware resources than [101]'s design with an exception of BRAMs and DSP blocks. To achieve such high performance, [101] employs manual deep-pipelining and symmetry of the A matrix is exploited. Their dense linear solver is able to handle matrix order of up to $N = 145$ while our design is estimated up to $N = 64$. Such large matrix implementation is possible due to extensive re-use of floating-point square root and division operators. However, we did not implement such design optimization strategy to avoid disrupting the structural regularity as it may conflict with the parametrisable nature and operations of our proposed TSA-based linear solver. In addition, such design optimizations would increase the design complexity of the Control Unit module, within the respective PEs, and increases the chances of requiring an experienced hardware designer to construct our proposed linear solver. The effects of design complexity trade-off can be considered as part of future work.

6.2.3 Performance and Resource Scalability

For the purpose of comparison with similar work, the solvers were implemented for a problem size of up to $N = 16$. To obtain an estimate for problem sizes up to $N = 128$, the Power Regression Model method was used and the results were plotted using Microsoft Excel. To explain, the trend-lines for FPGA Hardware Slices and hardware design frequency were plotted against varying matrix sizes, Figures 6.5 and Figures 6.6 respectively. The corresponding graph equations are derived and the hardware slices and design frequency were estimated for matrix problem size from $N = 32$ to $N = 128$. Thereafter, the estimated results were consolidated, together with existing data, into a single graph for the purpose of illustrating the trade-off between hardware performance and resource across different problem sizes, shown in Figures 6.7 and 6.8. In both figures, the solid lines refers to actual data points while the dotted lines are results estimaed using the Power Regression Model. From Figures 6.7 and 6.8, the solver design trade-off appears to be consistent for problem size of N>64.

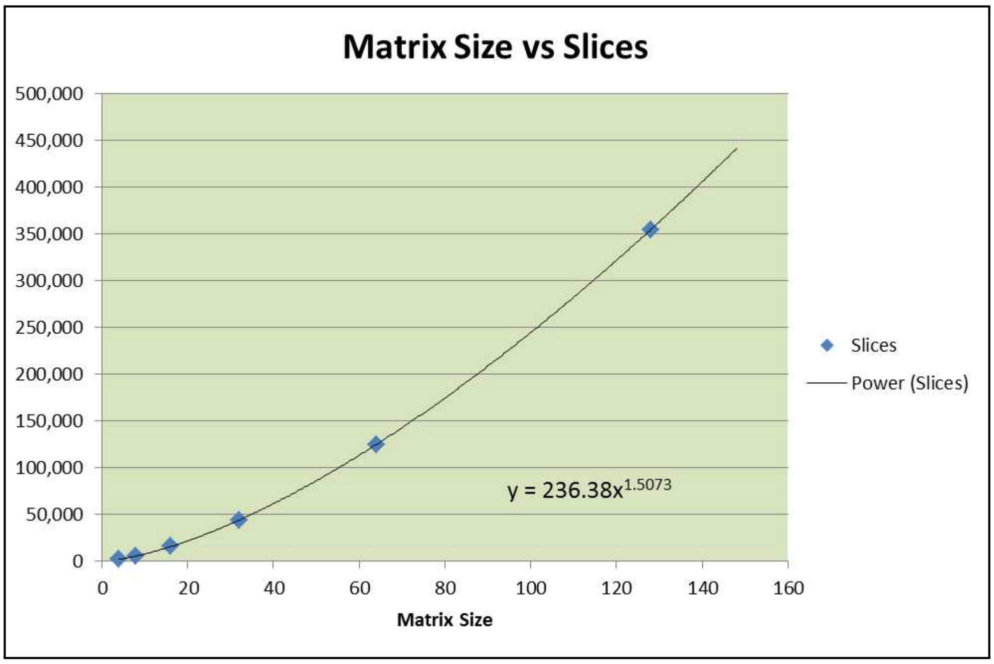


Figure 6.5: Regression Model Results for Proposed TSA-based Linear Solver (Matrix Size vs Slices)

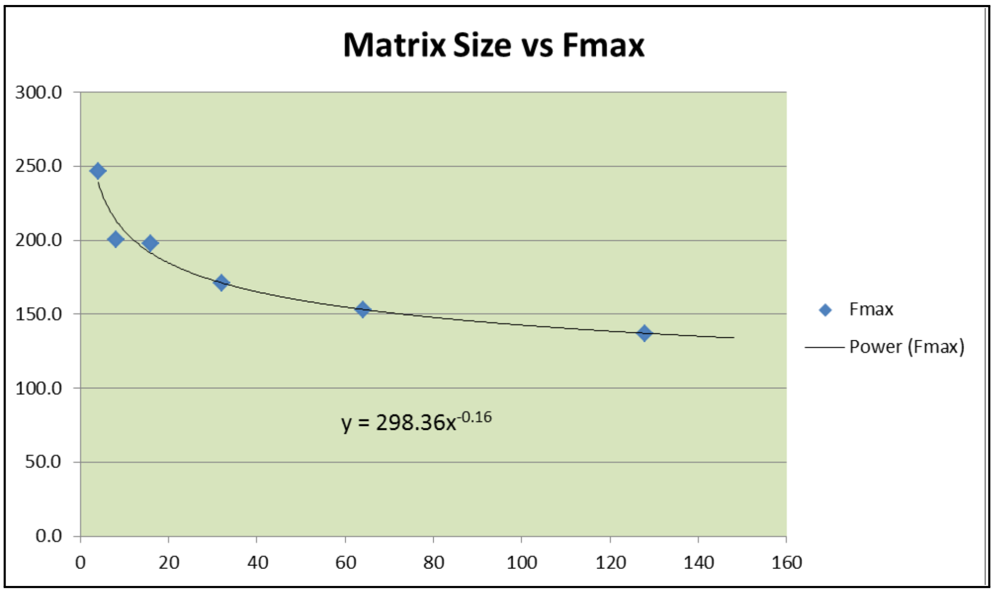


Figure 6.6: Regression Model Results for Proposed TSA-based Linear Solver (Matrix Size vs Fmax)

As previously mentioned, structural regularity of TSA enables a scalable hardware architecture design with deterministic hardware resources. For example, to scale

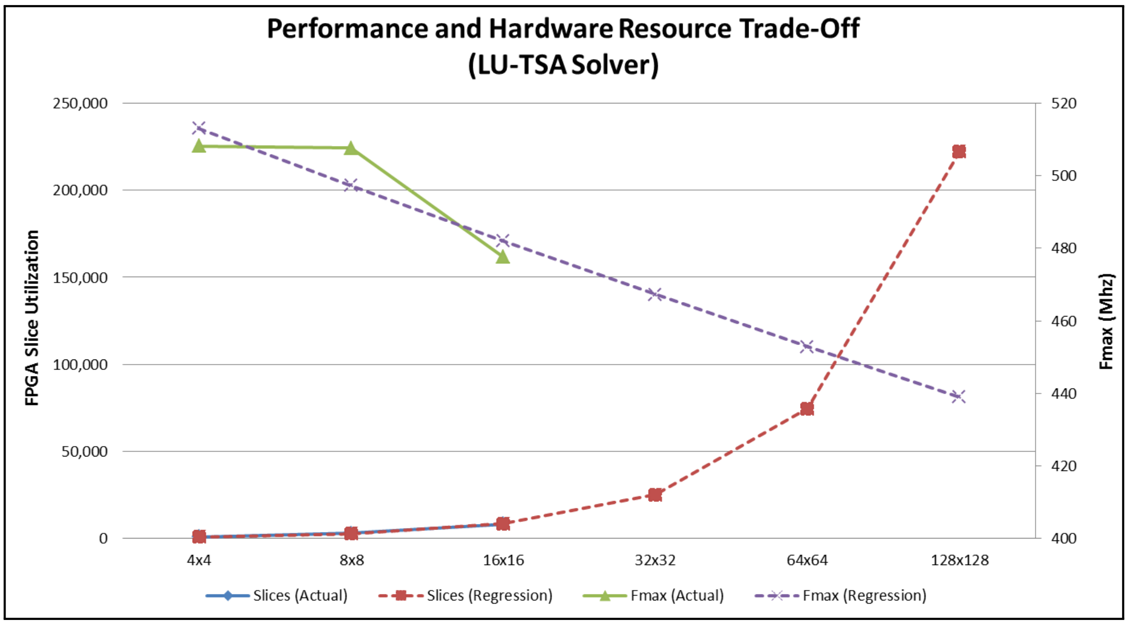


Figure 6.7: Hardware Performance and Resource Trade-Off for LU-TSA

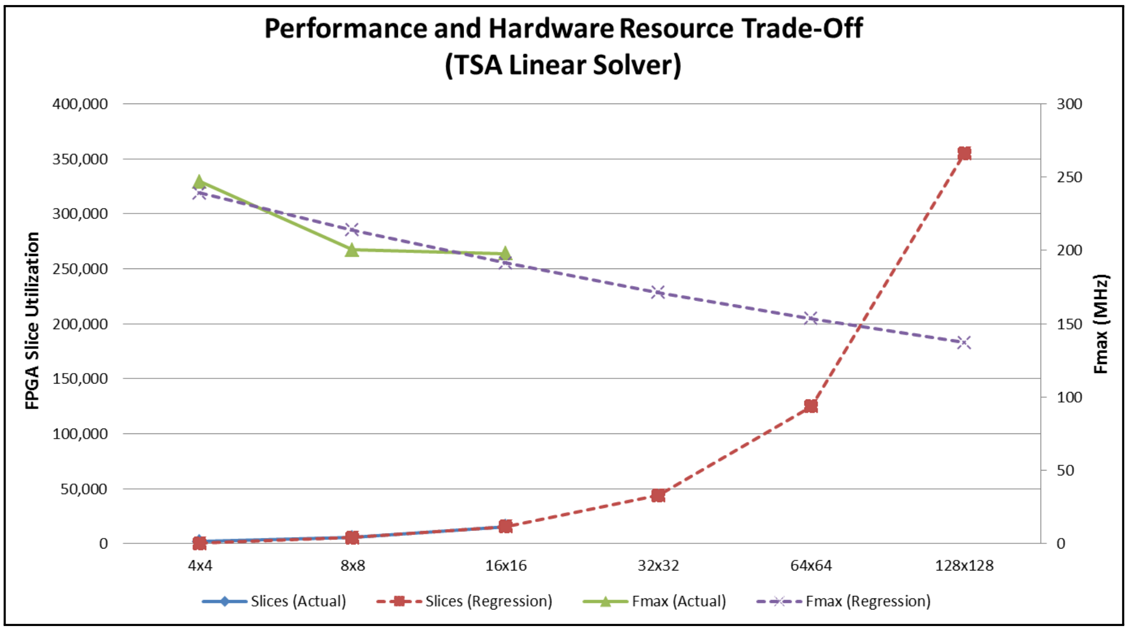


Figure 6.8: Hardware Performance and Resource Trade-Off for Proposed TSA-based Linear Solver

from a 4 x 4 to 8 x 8, users can quickly determine the number and additional quantity of PEs to be included in the design using the following formula:

- TSA: $\left[\frac{N(N+1)}{2} - 1\right]$ (see Chapter 5)

- Data Router: $N - 1$
- Data Arbiter: 1

Once the values of $N = 3$ and $N = 8$ are substituted into the formula, the required number of PEs and the PE types can be quickly determined by users. Hence, the designer only need instantiate the respective PE types. Figure 6.9 shows the number of PEs required for TSA design where $N = 8$ and the number of PEs for $N = 3$ is within the shaded area in Figure 6.9. The same concept is also applicable to the proposed TSA linear solver architecture.

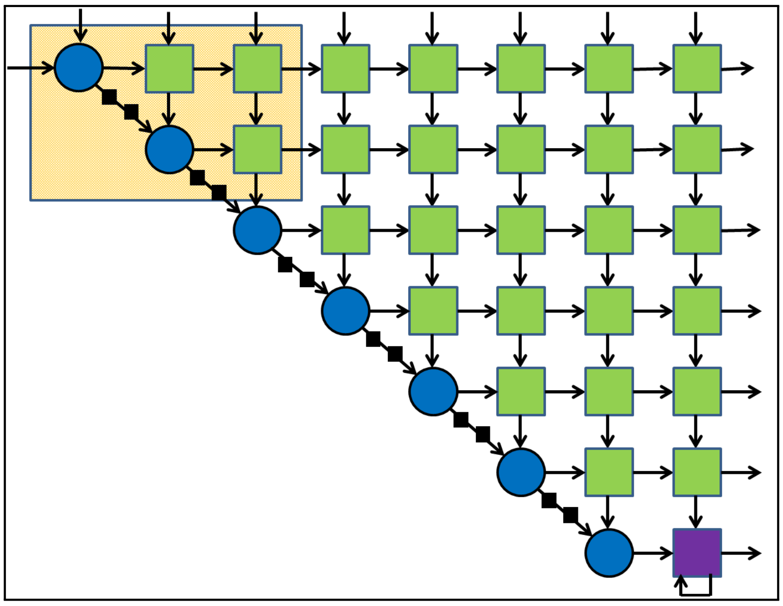


Figure 6.9: Design Scalability Example for TSA from $N = 3$ to $N = 8$

6.2.4 Summary

We would like to point out that majority of existing hardware solver design are point based solutions or application-specific, i.e they have been designed for one specific set of design parameters. Hence, a digital circuit designer is required if the design needs to be modified. On the other hand, our proposed TSA-based approach has structural regularity and scales easily according to the design problem size without the need for digital circuit designers. As a result, non-circuit designer can utilize fast LU solver or Linear Solver for their scientific application and only need focus on the architecture level. As previously explained, our proposed solvers (LU and Linear) are at least a few orders of magnitude faster than similar work whilst requiring up to 50% less hardware resources for the same problem size.

From the power consumption perspective, actual processing utilization for our TSA-based SA design maybe comparable to a 1D SA design. Our proposed TSA-based Linear Solver can also exhibit 1D SA like power consumption through the use of clock-gating on FPGAs, which can turn on and off PEs according to their usage.

Chapter 7

Conclusion

A systolic array based linear solver has been presented and implemented on an FPGA platform. Unlike previously reported work, the proposed design architecture does not side-step the computationally expensive floating-point division operations yet requires 2N less time-steps. We have also explained and demonstrated that careful exploitation of idle sequential cycles, floating-point divider block and omission of redundant arithmetic operations enabled the novel properties of the proposed systolic array based linear solver. Our proposed linear solver has a throughput of approximately 1 million linear systems for matrices of size $N = 4$ and approximately 82 thousand linear systems for matrices of size $N = 16$ respectively. In comparison with similar work, the proposed design offers up to a 12x improvement in speed whilst requiring up to 50% less hardware resources. As a result, the proposed linear solver design can be implemented for up to $N = 64$ on the largest Virtex 7 FPGA, which was previously not possible. Despite knowledge of the design trade-off, further investigation is required to validate the linear assumptions for problem sizes of $N \geq 32$. In addition, claims of our proposed Linear Solver exhibiting 1D Systolic Array like power consumption requires validation and is also part of the future work.

The key advantage of our design approach is that it empowers a non-circuit designer to utilize a fast LU solver or Linear Solver for their scientific application and they only need to focus on the architecture level. The ease of designing a scalable linear solver using systolic array approach has been demonstrated, prototyped and validated using Xilinx System Generator software tool. Due to the limited time, design automation for generation of the linear solver design, in accordance to user-defined problem size (N), was not investigated and will be included as future work.

The proposed systolic array design did not exploit special properties of the A matrix, such as symmetry, banded and sparse data, and this will be left as future work. Secondly, we intend to increase the number of design parameterization options as our current linear solver is only word length parameterizable with possible design automation of existing linear solver design based on user-defined problem size parameter. Fourthly, hardware constraints of FPGA resources limits the problem size our linear solver is able to achieve and we plan to look into alternative ways to decompose large problem sizes (i.e ≥ 128) into smaller problem sizes with multiplexed MPC as a potential candidate.

The design of the Data Arbiter and Data Router are currently external to the last column of PEs. Part of the future work can include the design integration and refinements into the last column of Mult-Sub/Modified-Mult-Sub PEs to further reduce the non-circuit designer′s developmental efforts. In addition, internal test result indicates that there is still room for performance improvement for the Data Arbiter and Data Router PEs and this will also be left as future work.

In this thesis, we assumed the SysGen generated linear solver design is efficient with little or no translation overhead. In future, we plan to quantify this translation overhead by design comparison with a hand-coded HDL implementation. In addition, although our proposed Hybrid, fixed-point and floating-point, numerical word-length precision have been adopted we did not highlight the benefits and motivation for such an adoption. In future, we plan to conduct a performance and hardware resource trade-off comparison between fixed-point versus Hybrid versus floating-point.

Lastly, we would like to reminder readers that the proposed compact scalable systolic array architecture is not constrained to MPC applications and can be applied to general scientific computing problems where a system of linear equations requires to be solved.

Bibliography

[1] T. J. Todman, G. A. Constantinides, S. J. Wilton, O. Mencer, W. Luk, and P. Y. Cheung, “Reconfigurable computing: architectures and design methods,” *IEE Proceedings-Computers and Digital Techniques*, vol. 152, no. 2, pp. 193–207, 2005.

[2] J. M. Cardoso, P. C. Diniz, and M. Weinhardt, “Compiling for reconfigurable computing: A survey,” *ACM Computing Surveys (CSUR)*, vol. 42, no. 4, p. 13, 2010.

[3] P. Orukpe, “Basics of model predictive control,” *Imperial College, London*, 2005.

[4] S. Y. Kung, “VLSI array processors,” *Englewood Cliffs, NJ, Prentice Hall, 1988, 685 p. Research supported by the Semiconductor Research Corp., SDIO, NSF, and US Navy.*, vol. 1, 1988.

[5] Z. Matej, “Systolic Parallel Processing Notes,” *https://ldos.fe.uni-lj.si/slo/03_Lectures/, retrieved online on 25/03/2013*, 2011.

[6] B. Inc., “Bluespec Training Slides,” *http://bluespec.com/forum/download.php ?id=106, retrieved online on 11-Feb-2012*, 2011.

[7] Intel, “Intel i7-3960X Extreme Processor,” *http://www.intel.com/content/ www/us/en/proce ssors/core/core-i7ee-processor.html, retrieved online on 14-Jan-2012*, 2012.

[8] AMD, “AMD Processors,” *http://www.amd.com, retrieved online on 14-Jan-2012*, 2012.

[9] D. Boland and G. A. Constantinides, “Automated precision analysis: A polynomial algebraic approach,” in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on.* IEEE, 2010, pp. 157–164.

[10] Xilinx, “Xilinx Zynq-7000 Manual,” *http://www.xilinx.com/products/silicon-devices/epp/zumq-7000/index.htm, retrieved online on 29/12/2011*, 2011.

[11] M. Wirthlin, D. Poznanovic, P. Sundararajan, A. Coppola, D. Pellerin, W. Najjar, R. Bruce, M. Babst, O. Pritchard, P. Palazzari *et al.*, “OpenFPGA CoreLib core library interoperability effort,” *Parallel Computing*, vol. 34, no. 4, pp. 231–244, 2008.

[12] National Instruments, “NI LabVIEW FPGA,” *http://www.ni.com/fpga/, retrieved online on 11-Feb-2012*, 2002.

[13] Mentor Graphics, “Handel-C Synthesis Methodology,” *http://www.mentor.com/products/fpga/handel-c/, retrieved online on 14-Jan-2012*, 2012.

[14] Impulse Accelerated Technologies, “ImpulseC Software,” *http://www. impulseaccelerated.com/, retrieved online on 14-Jan-2012*, 2012.

[15] Xilinx, “Xilinx EDK Platform,” *www.xilinx.com/tools/platform.htm, retrieved online on 14-Jan-2012*, 2012.

[16] Xilinx and Mathworks, “MATLAB Simulink HDL Coder,” *http://www.mathworks.com/products/slhdlcoder/, retrieved online on 14-Jan-2012*, 2012.

[17] Synopsys, “Synopsys Synphony Model Compiler,” *http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/default.aspx, retrieved online on 01/08/2014*, 2014.

[18] Xilinx, “Vivado,” *http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/, retrieved online on 01/08/2014*, 2014.

[19] K.-V. Ling, B. F. Wu, and J. Maciejowski, “Embedded model predictive control (MPC) using a FPGA,” in *Proc. 17th IFAC World Congress*, 2008, pp. 15 250–15 255.

[20] Xilinx, “Xilinx CORE Generator System,” *http://www.xilinx.com/tools/ coregen.htm, retrieved online on 20-Mar-2014*, 2014.

[21] Altera, “Altera MegaCore Functions,” *http://www.altera.com/products/ip/design/ipm-design.html, retrieved online on 20-Mar-2014*, 2014.

[22] OpenCores, “OpenCores,” *http://opencores.org/, retrieved online on 14-Jan-2012*, 2012.

[23] Xilinx, *System Generator for DSP User Guide UG640 (v 14.3).* Xilinx, 2012.

[24] Altera, “Altera DSP Builder,” *http://www.altera.com/products/software /products/dsp/dsp-builder.html, retrieved online on 14-Jan-2012*, 2012.

[25] Ong, Kevin SH, Suhaib A. Fahmy, and Keck-Voon Ling, "A Scalable and Compact Systolic Architecture for Linear Solvers," *Poster in Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), Zurich, Switzerland, June 2014*, pp. 186–187.

[26] J. L. Jerez, G. A. Constantinides, and E. C. Kerrigan, "An FPGA implementation of a sparse quadratic programming solver for constrained predictive control," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays.* ACM, 2011, pp. 209–218.

[27] M. S. Lau, S. P. Yue, K. V. Ling, and J. Maciejowski, "A comparison of interior point and active set methods for FPGA implementation of model predictive control," in *Proc. European Control Conference*, 2009, pp. 156–160.

[28] S. J. Wright, "Applying new optimization algorithms to model predictive control," in *AIChE Symposium Series*, vol. 93, no. 316. Citeseer, 1997, pp. 147–155.

[29] J. Maciejowski, "Predictive control with constraints," *Harlow, England: Pearson Education*, 2002.

[30] S. J. Wright, "Interior point methods for optimal control of discrete time systems," *Journal of Optimization Theory and Applications*, vol. 77, no. 1, pp. 161–187, 1993.

[31] J. E. Volder, "The CORDIC trigonometric computing technique," *Electronic Computers, IRE Transactions on*, no. 3, pp. 330–334, 1959.

[32] W. M. Gentleman and H. Kung, "Matrix triangularization by systolic arrays," in *25th Annual Technical Symposium.* International Society for Optics and Photonics, 1982, pp. 19–26.

[33] R. Woods, J. McAllister, G. Lightbody, and Y. Yi, *FPGA-based Implementation of Signal Processing Systems.* Wiley Online Library, 2008.

[34] M. Karkooti, J. R. Cavallaro, and C. Dick, "FPGA implementation of matrix inversion using QRD-RLS algorithm," in *Proceedings of the 39th Asilomar Conference on Signals, Systems and Computers*, 2005, pp. 1625–1629.

[35] Y.-W. Huang, C.-Y. Chen, C.-H. Tsai, C.-F. Shen, and L.-G. Chen, "Survey on block matching motion estimation algorithms and architectures with new results," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 42, no. 3, pp. 297–320, 2006.

[36] X. Wang and M. Leeser, "A truly two-dimensional systolic array FPGA implementation of QR decomposition," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, no. 1, p. 3, 2009.

[37] J. Moreno and T. Lang, *Matrix Computations on Systolic-Type Arrays.* Springer, 1992, vol. 174.

[38] S. Kestur, J. D. Davis, and O. Williams, "BLAS Comparison on FPGA, CPU and GPU," in *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on.* IEEE, 2010, pp. 288–293.

[39] Bluespec Inc., "Bluespec SystemVerilog Language," *http://www.bluespec.com, retrieved online on 29/12/2011*, 2011.

[40] R. S. Nikhil *et al.*, "What is Bluespec?" *ACM SIGDA Newsletter*, vol. 39, no. 1, pp. 1–1, 2009.

[41] C. C. Lin, "Implementation of H. 264 Decoder in Bluespec SystemVerilog," Ph.D. dissertation, Massachusetts Institute of Technology, 2007.

[42] A. Agarwal, M. C. Ng *et al.*, "A Comparative Evaluation of High-Level Hardware Synthesis Using Reed–Solomon Decoder," *Embedded Systems Letters, IEEE*, vol. 2, no. 3, pp. 72–76, 2010.

[43] Altera, "ModelSim-Altera Software," *http://www.altera.com/products/software/quartus-ii/modelsim/qts-modelsim-index.html, retrieved online on 14-Jan-2012*, 2012.

[44] K. V. Ling, S. P. Yue, and J. Maciejowski, "An FPGA Implementation of Model Predictive Control," *In Proc. American Control Conference*, pp. 1930–1935, 2006.

[45] G. A. Constantinides, "Tutorial paper: Parallel architectures for model predictive control," in *Proceedings of the European Control Conference, Budapest*, 2009, pp. 138–143.

[46] W. P. Marnane, C. Jordan, and F. O'Reilly, "Compiling regular arrays onto FPGAs," in *Field-Programmable Logic and Applications.* Springer, 1995, pp. 178–187.

[47] S. Stark and A. N. Beris, "LU decomposition optimized for a parallel computer with a hierarchical distributed memory," *Parallel computing*, vol. 18, no. 9, pp. 959–971, 1992.

[48] M. C. Chen, "Placement and interconnection of systolic processing elements: A new LU-decomposition algorithm," in *Proc. IEEE Int. Conf. on Computer Design (ICCD86)*, 1986, pp. 275–281.

[49] M. Mosleh, S. Setayeshi, and M. Kheyrandish, "Presenting a Systematic Method for LU Decomposition of a Matrix with Linear Systolic Arrays," in *Advanced Computer Theory and Engineering, 2008. ICACTE'08. International Conference on.* IEEE, 2008, pp. 123–127.

[50] A. Irturk, J. Matai, J. Oberg, J. Su, and R. Kastner, "Simulate and eliminate: A top-to-bottom design methodology for automatic generation of application specific architectures," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 8, pp. 1173–1183, 2011.

[51] A. U. Irturk, "GUSTO: General architecture design Utility and Synthesis Tool for Optimization," Doctoral thesis, University Of California, San Diego, 2009.

[52] S. Choi and V. K. Prasanna, "Time and energy efficient matrix factorization using FPGAs," in *Field Programmable Logic and Application*. Springer, 2003, pp. 507–519.

[53] M. K. Jaiswal and N. Chandrachoodan, "FPGA-based high-performance and scalable block LU decomposition architecture," *Computers, IEEE Transactions on*, vol. 61, no. 1, pp. 60–72, 2012.

[54] A. C. R.Clint Whaley, "Automatically Tuned Linear Algebra Software (ATLAS)," *http://math-atlas.sourceforge.net/, retrieved online on 25/03/2014*, 2011.

[55] Intel Corporation, "Math Kernel Library (MKL)," *http://software.intel.com/en-us/intel-mkl, retrieved online on 25/03/2014*, 2011.

[56] B. Fang, S. Chen, and X. Wei, "Single-precision LU decomposition based on FPGA compared with CPU," in *Computational Problem-Solving (ICCP), 2012 International Conference on*. IEEE, 2012, pp. 302–305.

[57] G. Wu, Y. Dou, J. Sun, and G. D. Peterson, "A high performance and memory efficient LU decomposer on FPGAs," *Computers, IEEE Transactions on*, vol. 61, no. 3, pp. 366–378, 2012.

[58] G. Wu, Y. Dou, and G. D. Peterson, "Blocking LU decomposition for FPGAs," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. IEEE, 2010, pp. 109–112.

[59] W. Zhang, V. Betz, and J. Rose, "Portable and scalable FPGA-based acceleration of a direct linear system solver," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 5, no. 1, p. 6, 2012.

[60] J. Humberto and G. Romero, "A comparative study of two wavefront implementations of a LU solver algorithm," in *CONPAR 90VAPP IV*. Springer, 1990, pp. 672–681.

[61] G. Wu, Y. Dou, Y. Lei, J. Zhou, M. Wang, and J. Jiang, "A fine-grained pipelined implementation of the LINPACK benchmark on FPGAs," in *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*. IEEE, 2009, pp. 183–190.

[62] J. J. Dongarra, “The linpack benchmark: An explanation,” in *Supercomputing.* Springer, 1988, pp. 456–474.

[63] C. Wan, “Systolic algorithms and applications,” Doctoral thesis, University of Loughborough University, 1996.

[64] Y.-G. Tai, C.-T. Dan Lo, and K. Psarris, “Scalable matrix decompositions with multiple cores on FPGAs,” *Microprocessors and Microsystems*, 2012.

[65] Y. Dou, J. Zhou, G.-M. Wu, J.-F. Jiang, Y.-W. Lei, and S.-C. Ni, “A unified co-processor architecture for matrix decomposition,” *Journal of Computer Science and Technology*, vol. 25, no. 4, pp. 874–885, 2010.

[66] M. Zubair and B. Madan, “Efficient systolic structures for LU decomposition and system of linear equations,” *Circuits, Systems and Signal Processing*, vol. 7, no. 2, pp. 275–287, 1988.

[67] I. Bravo, P. Jimenez, M. Mazo, J. L. Lazaro, J. J. de las Heras, and A. Gardel, “Different proposals to matrix multiplication based on FPGAs,” in *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on.* IEEE, 2007, pp. 1709–1714.

[68] E. Casseau and D. Degrugillier, “A linear systolic array for LU decomposition,” in *VLSI Design, 1994., Proceedings of the Seventh International Conference on.* IEEE, 1994, pp. 353–358.

[69] D. Kim and S. V. Rajopadhye, “An improved systolic architecture for LU decomposition,” in *Application-specific Systems, Architectures and Processors, 2006. ASAP’06. International Conference on.* IEEE, 2006, pp. 231–238.

[70] G. Valencia-Palomo, K. Hilton, and J. Rossiter, “Predictive control implementation in a PLC using the IEC 1131.3 programming standard,” in *Proceedings of The European Control Conference 2009*, 2008, pp. 23–26.

[71] G. Valencia-Palomo and J. Rossiter, “Programmable logic controller implementation of an auto-tuned predictive control based on minimal plant information,” *ISA transactions*, vol. 50, no. 1, pp. 92–100, 2011.

[72] A. Syaichu-Rohman and R. Sirius, “Model predictive control implementation on a programmable logic controller for DC motor speed control,” in *Electrical Engineering and Informatics (ICEEI), 2011 International Conference on.* IEEE, 2011, pp. 1–4.

[73] M. Mrosko and E. Miklovičová, “Real-time implementation of predictive control using programmable logic controllers,” *International Journal of Systems Applications, Engineering & Development Issue 1*, vol. 6, 2012.

[74] L. G. Bleris and M. V. Kothare, “Real-time implementation of model predictive control,” in *American Control Conference, 2005. Proceedings of the 2005.* IEEE, 2005, pp. 4166–4171.

[75] A. K. Abbes, F. Bouani, and M. Ksouri, “A microcontroller implementation of constrained model predictive control,” *IJ Electr. Electron. Eng*, vol. 5, no. 3, pp. 199–206, 2006.

[76] P. Zometa, M. Kogel, T. Faulwasser, and R. Findeisen, “Implementation aspects of model predictive control for embedded systems,” in *American Control Conference (ACC), 2012.* IEEE, 2012, pp. 1205–1210.

[77] NVIDIA, “Compute Unified Device Architecture (CUDA) Parallel Computing Platform,” *http://www.nvidia.com, retrieved online on 25/03/2014*, 2014.

[78] T. K. Group, “OpenCL,” *https://www.khronos.org/opencl/, retrieved online on 25/03/2014*, 2014.

[79] Y. Huang, K. V. Ling, and S. See, “Solving Quadratic Programming Problems On Graphics Processing Unit,” *ASEAN Engineering Journal 2011*, vol. 1, no. 2.

[80] K. Turkington, G. A. Constantinides, K. Masselos, and P. Y. Cheung, “Outer loop pipelining for application specific datapaths in FPGAs,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 10, pp. 1268–1280, 2008.

[81] A. R. Lopes and G. A. Constantinides, “A high throughput FPGA-based floating point conjugate gradient implementation,” in *Reconfigurable Computing: Architectures, Tools and Applications.* Springer, 2008, pp. 75–86.

[82] D. Boland and G. A. Constantinides, “An FPGA-based implementation of the MINRES algorithm,” in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on.* IEEE, 2008, pp. 379–384.

[83] A. Shahzad, E. C. Kerrigan, and G. A. Constantinides, “Preconditioners for inexact interior point methods for predictive control,” in *American Control Conference (ACC), 2010.* IEEE, 2010, pp. 5714–5719.

[84] J. Currie and D. I. Wilson, “A Model Predictive Control toolbox intended for rapid prototyping,” in *16th Electronics New Zealand Conference (ENZCon 2009)*, 2009, pp. 7–12.

[85] J. Currie and D. Wilson, “Lightweight model predictive control intended for embedded applications,” in *The 9th International Symposium on Dynamics and Control of Process Systems, Leuven, Belgium*, 2010.

[86] D. Boland and G. A. Constantinides, "Optimizing memory bandwidth use and performance for matrix-vector multiplication in iterative methods," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 3, p. 22, 2011.

[87] M. He and K. V. Ling, "Model predictive control on a chip," in *Control and Automation, 2005. ICCA'05. International Conference on*, vol. 1. IEEE, 2005, pp. 528–532.

[88] G. M. de Matos and H. C. Neto, "On reconfigurable architectures for efficient matrix inversion," in *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on.* IEEE, 2006, pp. 1–6.

[89] A. Wills, G. Knagge, and B. Ninness, "Fast linear model predictive control via custom integrated circuit architecture," *Control Systems Technology, IEEE Transactions on*, vol. 20, no. 1, pp. 59–71, 2012.

[90] G. Knagge, A. Wills, A. Mills, and B. Ninness, "ASIC and FPGA implementation strategies for model predictive control," in *Proc. European Control Conference*, 2009.

[91] A. Wills, A. Mills, and B. Ninness, "FPGA Implementation of an Interior-Point Solution for Linear Model Predictive Control," in *Preprints of the 18th IFAC World Congress, Milano, Italy*, 2011, pp. 14 527–14 532.

[92] J. L. Jerez, G. A. Constantinides, and E. C. Kerrigan, "FPGA implementation of an interior point solver for linear model predictive control," in *Field-Programmable Technology (FPT), 2010 International Conference on.* IEEE, 2010, pp. 316–319.

[93] A. R. Lopes, A. Shahzad, G. A. Constantinides, and E. C. Kerrigan, "More flops or more precision? Accuracy parameterizable linear equation solvers for model predictive control," in *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on.* IEEE, 2009, pp. 209–216.

[94] P. D. Vouzis, L. G. Bleris, M. G. Arnold, and M. V. Kothare, "A custom-made algorithm-specific processor for model predictive control," in *Industrial Electronics, 2006 IEEE International Symposium on*, vol. 1. IEEE, 2006, pp. 228–233.

[95] L. G. Bleris, P. D. Vouzis, M. G. Arnold, and M. V. Kothare, "A co-processor FPGA platform for the implementation of real-time model predictive control," in *American Control Conference, 2006.* IEEE, 2006, pp. 6–pp.

[96] K. S. L. Christopher, "Solving Interior Point Method on FPGA," Masters thesis, Nanyang Technological University Singapore, 2009.

[97] A. Mills, A. Wills, S. Weller, and B. Ninness, “Implementation of linear model predictive control using a field-programmable gate array,” *IET control theory & applications*, vol. 6, no. 8, pp. 1042–1054, 2012.

[98] J. Jerez, K.-V. Ling, G. Constantinides, and E. Kerrigan, “Model predictive control for deeply pipelined field-programmable gate array implementation: algorithms and circuitry,” *Control Theory & Applications, IET*, vol. 6, no. 8, pp. 1029–1041, 2012.

[99] L. G. Bleris, M. V. Kothare, J. Garcia, and M. G. Arnold, “Embedded model predictive control for system-on-a-chip applications,” *Proceedings of the 7th IFAC Symposium on Dynamics and Control of Process Systems (DYCOPS-7)*, 2004.

[100] Y. H. Hu and S.-Y. Kung, “Systolic arrays,” in *Handbook of Signal Processing Systems.* Springer, 2013, pp. 1111–1143.

[101] D. Boland and G. A. Constantinides, “Optimizing memory bandwidth use and performance for matrix-vector multiplication in iterative methods,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 3, p. 22, 2011.