

ABSTRACT

DIWAN, MAITRIK Methodology for Analyzing Complex Algorithms for Small Satellites. (Under the direction of Professor Willam W. Edmonson).

From the view point of an electrical engineer, one of the main challenges in the field of space science is the mapping of complex control algorithms onto appropriate hardware architectures. There is a wide knowledge gap between the team which designs these algorithms and the team which implements the algorithms. There is a variety of hardware architectures available commercially onto which the algorithms can be mapped. There also exist different design tools which can be used to perform the implementation of these complex algorithms on appropriate hardware architectures. With the availability of a large suite of architectures and design tools, the design engineer often gets puzzled in selecting the correct architecture and tools to perform the implementation of the algorithm in an optimal manner.

This thesis presents a methodology for analyzing complex algorithms for small satellite applications. The proposed methodology provides a step by step process of analyzing an algorithm for converting its Platform Independent Model (PIM) into a Platform Specific Model (PSM). It assumes that the algorithm has been designed in Simulink from Mathworks, which is the design tool of choice for aerospace system engineers. After this, the methodology describes the detailed process of performing complexity analysis of the algorithm, which leads to identification of the accelerator components. The next step explains the process of software and hardware implementation of the accelerator components. After this, the software and hardware implementations are compared on the basis of Speed of Execution, Power Dissipation and Development Time. The results are analyzed to make a decision regarding the implementation of the algorithm.

The proposed methodology is validated by applying it to the Attitude Determination and Control algorithm designed for attitude control of small satellites.

Methodology for Analyzing Complex Algorithms for Small Satellites

by

Maitrik Diwan

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Electrical and Computer Engineering

Raleigh, North Carolina

2007

Approved By:

Dr. Winsor E. Alexander

Dr. William Rhett Davis

Dr. William W. Edmonson
Chair of Advisory Committee

Dedication

To

Mom, Dad and Prerak

Biography

Maitrik Diwan was born on December 25, 1983 in Ahmedabad, a city in the western part of India. He received his Bachelor of Engineering (B.E.) degree in Electronics and Communication from Nirma Institute of Technology (Gujarat University) in June 2005. He worked at the Physical Research Laboratory (A unit of Department of Space, Government of India), Ahmedabad as a System Design Engineer for his senior year project.

Maitrik joined NC State University, Raleigh in Fall 2005 to pursue graduate study in Computer Engineering. Since Spring 2006, he has been working with Dr. William Edmonson in the field of design and development of small satellites. He has been a part of a collaborative space based project between the North Carolina State University, the University of Florida, Gainesville, and the Defense Advanced Research Projects Agency (DARPA). His research interests include Hardware/Software Co-Design and Asic Verification.

Acknowledgements

Above all, I thank my parents Jayshree Diwan and Yogesh Diwan for the much needed motivation throughout the duration of this project. It was their love and support that helped me maintain sanity during stressful times. I would also like to thank my dearest brother Prerak Diwan for his ever lasting support.

I sincerely acknowledge the efforts of Dr. William Edmonson, my academic advisor, in providing guidance and encouragement for the successful completion of this thesis. Dr. Edmonson has made available all resources that I could possibly need and also allowed the independence of applying my ideas in this project. I am deeply indebted to him for his patience and invaluable suggestions during the course of this project. I am also grateful to the members of my thesis committee, Dr. Winsor Alexander and Dr. Rhett Davis for devoting their time and providing useful inputs.

I am highly grateful to all the members of HiperDSP Lab at NC State for providing their support and encouragement during the course of my thesis. Special thanks to Ramsey Hourani and Young Soo Kim, without whose support, my stay at HiperDSP lab would have been nearly impossible. They have always pointed me to the true direction of research during the course of my thesis. Once again, I sincerely thank Ramsey Hourani for all his academic as well as non-academic support. I would also like to thank Ravi Jenkal for his valuable suggestions throughout the course of my research.

I would also like to acknowledge the engineers at Xilinx Inc., *Mathworks*® and Texas Instruments for providing technical support for the successful completion of my thesis.

Finally, I would like to thank my roommates- Kaushal, Gunjan, and Parth, and my labmate- Sandeep for their support and making my life at NC State memorable.

Contents

List of Figures	vii
List of Tables	ix
1 Introduction	1
1.1 Motivation	4
1.2 Related Work	6
1.3 Contribution	8
1.4 Thesis Organization	9
2 Design Space Exploration	10
2.1 Choice of Hardware Architecture	11
2.1.1 Time to Launch	11
2.1.2 Performance	11
2.1.3 Price	12
2.1.4 Power Dissipation	12
2.1.5 Feature Flexibility	13
2.2 Architecture Exploration	13
2.2.1 General Purpose Processors (GPP/RISC)	14
2.2.2 Digital Signal Processors (DSP)	14
2.2.3 Field Programmable Gate Arrays (FPGA)	15
2.2.4 Application Specific Integrated Circuits (ASIC)	16
2.3 Implementation Techniques	17
2.3.1 Software Only Implementation	19
2.3.2 Hardware Only Implementation	21
2.3.3 Hardware-Software Co-Design	22
3 Design Methodology	25
3.1 Proposed Methodology	27

4	FIR Filter Case Study	55
4.1	Code Comparison	56
4.2	Software Code Optimization	59
4.2.1	Development flow to increase performance	60
4.2.2	Optimization techniques	62
4.2.3	Summary	68
5	Validation of Proposed Methodology	71
5.1	Overview of ADC Algorithm [1]	71
5.2	Analysis of ADC algorithm	75
5.2.1	Step 1: Choose candidate platforms	75
5.2.2	Step 2: Algorithm analysis	76
5.2.3	Step 3: Choose candidate accelerators	78
5.2.4	Step 4: HW/SW implementation of accelerators	81
5.2.5	Step 5: Comparative Metrics measurement	100
5.2.6	Step 6: Analyze the results	107
5.3	Conclusion	115
6	Conclusions and Future Work	117
6.1	Conclusions	117
6.2	Future Work	120
	Bibliography	121

List of Figures

1.1	Attitude Determination and Control system designed by AMAS	5
2.1	Flexibility vs. Efficiency Tradeoff [2]	18
2.2	Code Generation Process from Simulink Model [3]	20
2.3	A generic high level synthesis system [4]	22
2.4	Global Scheme for Codesign Methodology	24
3.1	Overall flow diagram of the Proposed Methodology	28
3.2	Concept Scoring Matrix	30
3.3	Algorithm Analysis Flow	32
3.4	Flow of data between DSP and FPGA	37
3.5	Simulink model for Time Complexity Analysis	38
3.6	Software Implemenation Process	40
3.7	Hardware Implemenation Process	42
3.8	FPGA Power Consumption Estimation using XPower	50
3.9	Decision Making Flow	52
4.1	Direct Form Implementation of a 6th order FIR filter	57
4.2	Organization of hand written and tool generated codes	59
4.3	Code Development Flow for TI C6xxx	61
4.4	Code demonstrating Forward Store Optimization	64
4.5	Code demonstrating Loop Unrolling	65
4.6	Code demonstrating Index Updating	67
4.7	Code demonstrating use of Intrinsics	67
4.8	FIR filter design space	70
4.9	Performance Improvement for FIR filter	70
5.1	4 SGCMG cluster in a pyramid configuration	72
5.2	High Level Block diagram of ADC algorithm	73
5.3	Complexity analysis of Satellite Model	77
5.4	Matrix multiplication code	78

5.5	Selection of Candidate Accelerators	80
5.6	Optimizations performed on C matrix multiplication code	84
5.7	Hardware Implementation of 3×3 matrix multiplication	86
5.8	Software implementation of LU decomposition	89
5.9	Hardware implementation of LU decomposition	94
5.10	State Diagram of FSM I	95
5.11	Hardware architecture of division module	97
5.12	Max_mul module	98
5.13	Sub module	99
5.14	Diagram to illustrate concept of power dissipation calculation	103
5.15	Execution Time analysis for Matrix Multiplication	108
5.16	Power Dissipation analysis for Matrix Multiplication	109
5.17	Execution Time – Power Dissipation curve	109
5.18	Execution Time analysis for LU decomposition	112
5.19	Power Dissipation analysis for LU decomposition	113
5.20	Execution Time – Power Dissipation curve	114

List of Tables

1.1	Satellite Classification on the basis of weight	2
2.1	Architectures and their Characteristics	16
3.1	Characteristics and their numerical values	29
3.2	Tabular representation for TI matrix system	29
3.3	Different Categories of Time Complexities	36
4.1	Differences between hand written and tool generated codes	60
4.2	Clock cycles and Speedup for different Optimizations	69
5.1	Time Complexity of blocks of Torque Generator	79
5.2	Time Complexity of blocks of Steering Logic	79
5.3	Design Space for Matrix Multiplication	83
5.4	Hardware Implementation results for matrix multiplication	87
5.5	Design Space for LU Decomposition	92
5.6	Hardware Implementation results for LU decomposition	100
5.7	Exection Times for Software Implemetations of Matrix Multiplication	101
5.8	Exection Times for Software Implemetations of LU decomposition	102
5.9	DSP power analysis for Matrix multiplication	105
5.10	DSP power analysis for LU decomposition	106
5.11	Code length comparison	107
6.1	Summary of Results	119

Chapter 1

Introduction

In the late 1980s, a new design approach for utilizing modern small satellites arose, and opened up a new class of space applications. One-of-a-kind, low-profile, low-cost satellites, funded primarily by the Department of Defense (DoD) Advanced Research Projects Agency, the Air Force Space Test Program, and university laboratories were built. The satellites were built to maximize the use of existing components, off-the-shelf technology and minimization of non-recurring developmental effort. The trend towards using small satellites continues today as evidenced by programs such as the NASA Small Explorer (SMEX) Program and Earth Science System Pathfinder (ESSP), among others. NASA, DOD, foreign, and commercial developers of space systems have all recently shown increased interest in small satellites as vehicles for science, technology demonstration, remote sensing, and communications [5].

Many terms are used to describe this rediscovered class of satellites, including SmallSat, Cheapsat, MicroSat, MiniSat, NanoSat and even PicoSat and FemtoSat. The US Defence Advanced Research Projects Agency refers to these as LightSats, the U.S. Naval Space Command as SPINSat's (Single Purpose Inexpensive Satellite Systems), and the U.S. Air Force as TACSat's (Tactical Satellites). Nevertheless, in recent years a general method of classifying satellites in terms of deployed mass has

been generally adopted. The boundaries of these classes are an indication of where launcher or cost tradeoffs are typically made, which is also why the mass is defined including fuel [6]. Table 1.1 shows the classification of satellites on the basis of their wet mass (mass including fuel).

Small satellites have been used in a number of successful space missions. One of the major developments in the history of small satellites is the evolution of the Cubesat kit. The Cubesat kit is designed to help the user to complete a successful small satellite mission in as short of time as possible and at low cost [7]. It has been used to accomplish a number of successful space missions. On June 9, 2007, the Space and Systems Development Laboratory (Department of Aeronautics and Astronautics at Stanford University) successfully launched their third BioLaunch mission, B07C using the Cubesat kit [8]. On May 21, 2007, Kysat completed the first integration of its CubeSat Kit-based KySat1 nanosatellite, and verified its target mass of under 1kg [1]. All of these and many more successful experiments show that the Cubesat kit has been proven to provide inexpensive and timely access to space for small payloads.

Table 1.1: Satellite Classification on the basis of weight

<i>Group Name</i>	<i>Wet Mass</i>
Large Satellite	>1000kg
Medium Sized Satellite	500-1000kg
Mini-satellite	100-500kg
Micro-Satellite	10-100kg
Nano-Satellite	1-10kg
Pico-Satellite	0.1-1kg
Femto-satellite	<0.1kg

All satellites have one or more controllers which control the operation of the satellite. They collect the data measured by the satellite, process it and send it to the ground station. At the same time, they receive commands from the ground station and perform the required tasks. The controlling element can be considered as the heart of any space based vehicle. It is a piece of hardware which can house a number of algorithms targeted to perform a wide variety of tasks. The Cubesat kit consists of a Flight Model (10cm x 10cm x 10cm) which has a RISC-MCU-based FM430 Flight Module as the main controlling element. There is also a provision for plugging in other modules which can be used to perform a number of tasks like power supply and heath management [7].

From the view point of an electrical engineer, one of the main challenges in the field of space science is the mapping of complex control algorithms onto appropriate hardware architectures. There is a wide knowledge gap between the team which designs these algorithms and the team which implements the algorithms. There is a variety of hardware architectures available commercially onto which the algorithms can be mapped. There also exist different design tools which can be used to perform the implementation of these complex algorithms on appropriate hardware architectures. Although the tools speed up the design process, they often prove to be inefficient. With the availability of a large suite of architectures and design tools, the design engineer often gets puzzled in selecting the correct architecture and tools to perform the implementation of the algorithm in an optimal manner.

There are a number of requirements which have to be met while implementing an algorithm on particular hardware architecture. These requirements depend upon the application for which the implementation is targeted. In the case of designs for small satellites, some of these requirements are a) The design should be able to meet real time constraints b) The design should have minimum power dissipation c) The design should be reconfigurable, and d) The design should be small in size.

This thesis presents a directed methodology for analyzing complex control algo-

rithms for small satellites. The methodology starts with the identification of the candidate hardware architectures for implementing the algorithm. After selecting appropriate hardware architecture(s), it provides a directed flow with decision making at several stages, and ultimately leads to a final design that meets all the requirements. The methodology will be validated by applying it to the complex Attitude Determination and Control [1] algorithm designed for the attitude control of small satellites.

1.1 Motivation

As the utility of satellites grows from a strictly military and scientific venture to a vital component of large industries such as telecommunications, a greater number of organizations are looking at available Earth-to-Orbit (ETO) launch vehicles. The strong desire to reduce launch costs and, thereby, increase access to space has spawned a number of studies to advance the field of aerospace technology [9]. A wide variety of algorithms are being designed for performing different tasks by the satellites in space. One of the common categories of space based algorithms is that of the Attitude Control algorithms. The attitude control algorithms are the algorithms that receive input from the sensors and calculate the appropriate actuator commands. These commands, whether for reaction wheels, jets, or other actuators, are intended to apply forces and torques to rotate or translate the spacecraft to the desired attitude. The attitude data, which is processed by the algorithms, comes from the attitude sensors, for example gyroscopes, sun sensors, rotary encoders or star trackers. The algorithm can be a very simple proportional control or a complex nonlinear estimator or many types in-between, depending on the mission requirements.

One such algorithm has been designed by the AMAS Laboratory at the Mechanical and Aerospace Engineering Department at the University of Florida, Gainesville. The algorithm has been developed to enable small satellites to perform autonomous in-space operations [1]. A simplified block diagram of the overall system is shown

in Figure 1.1. It is the controller in Figure 1.1 which houses the attitude control algorithm. Here IMU refers to *Inertial Measurement Unit*, a sensor placed at the center of mass of the system shown below.

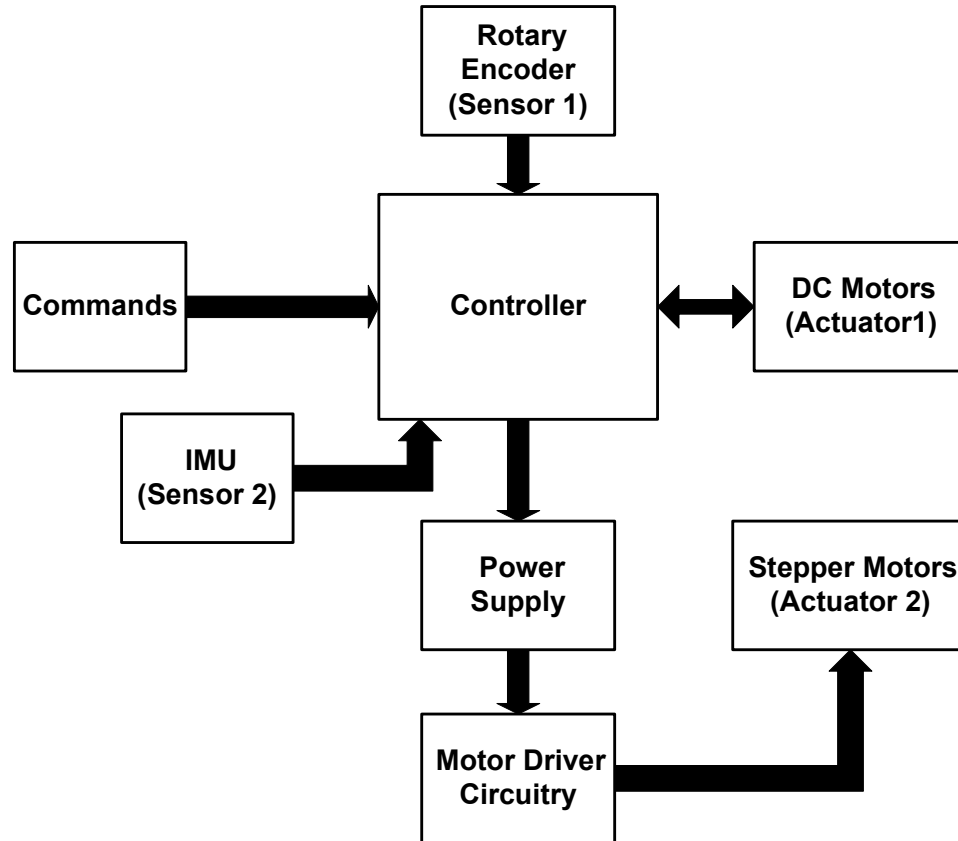


Figure 1.1: Attitude Determination and Control system designed by AMAS

The task of implementing this algorithm on hardware architecture has been undertaken by the HiperDSP Laboratory at the North Carolina State University, Raleigh. In the process of trying to implement the algorithm, it was realized that there was a need for a directed methodology which can guide a design engineer in taking the right decisions at all the levels starting from the selection of the right hardware architecture to the process of mapping the algorithm to a selected architecture.

The type of hardware architectures that the control algorithms can be mapped onto are General Purpose Processors (GPP), Digital Signal Processors (DSP), Field Programmable Gate Arrays (FPGA), Application Specific Instruction-Set Processors (ASIP), Reconfigurable Processors and Application Specific Integrated Circuits (ASIC) [10]. There are various suites of tools available for performing implementations on each of these architectures. The choice of the right set of tools is essential in any development process. It has been observed that tools from different vendors, which are part of a single development flow, may not be compatible with each other. It has also been observed that the same tool works well for one application model but does not work well for the other.

All the above factors lead to the idea of developing a methodology for implementing complex control algorithms on appropriate hardware architectures. The methodology which is described in this thesis can not only be used to analyze the Attitude Determination and Control [1] algorithm, but a number of other algorithms being developed at the AMAS Laboratory.

1.2 Related Work

A number of design based methodologies have been developed for various applications. All the methodologies start their flow with the Platform Independent Model [11] of the system. Using the commercially available tools, they analyze the complexity of the algorithms and take the decision regarding the architecture on which the algorithm can be implemented and the tools which can be used to implement the algorithms on the selected architectures. Some of the methodologies that have been developed so far are listed below:

- *An Efficient Methodology for Design and Verification of Software Defined Radio* [12]

In this methodology, the authors talk about a design flow which is divided into

three main steps. The flow, leading from system specifications down to implementation, goes across multiple abstraction levels. The first step is to build a platform independent model of the system. Next the model goes through the partitioning process to generate the Platform Specific Model. This Platform Specific model is then refined down to the RTL level for integration on the target board. The preliminary modeling and design of the system is done using Simulink from Mathworks. After the process of partitioning is carried out, the hardware part is implemented using VHDL and the software part is implemented using C/C++. The target board consists of FPGA and CPU core.

- *True DSP Synthesis* [13]

In this methodology, a Register Transfer Level (RTL) implementation of the algorithm is done automatically from the high level description of the algorithm. It is targeted mainly for DSP designs. It assumes that the implementation of the algorithm is being done on the FPGAs or the ASICs.

- *Mimosys Design Flow* [14]

In this design flow, the starting point is the C code of the application model. This C code is profiled by the inbuilt profiler of the tool. The profiling results identify the hardware accelerator components. After the components are identified, it generates the VHDL code for the hardware accelerators and modifies the original source code to make use of the newly generated hardware accelerator components. The modified source code and the HDL components are then imported to Xilinx Platform Studio [15] for the purpose of implementation on the FPGA.

Apart from the methodologies listed above, a number of other methodologies have been developed to implement algorithms on different hardware architectures. One of the main drawbacks of these methodologies is that they use certain tools which are not readily available. For example, one hardware-software codesign methodology described in [16] uses high level synthesis tools such as GAUT [17] and SYNDEX

[18]. GAUT is a high level synthesis tool for the ASICs and SYNDEX produces the dedicated code for the DSPs. These are the tools which are developed for the purpose of research and are not commercially available. Moreover, the methodologies that have been developed so far are targeted to specific class of applications and have been devised keeping in mind those specific applications. They cannot be applied to all the applications. All the above drawbacks lead to the process of development of a methodology for analyzing complex algorithms.

1.3 Contribution

The following thesis presents a methodology for analyzing complex control algorithms for small satellites. It starts with the selection of the right hardware architecture(s) and provides a directed flow involving decision making at several steps in order to reach a final conclusion. The methodology assumes that the given application model has been designed in Simulink by *Mathworks*[®]. An important decision making point in the methodology is whether to manually write the software description of the model or to use commercially available tools for the same purpose. A simple case study of an FIR filter is carried out to obtain a conclusion regarding the above mentioned point. The case study also describes the different optimization techniques that can be performed on the software description of a model. The proposed methodology is validated by applying it to the Attitude Determination and Control algorithm designed for the attitude control of small satellites. The complexity analysis of the algorithm is carried out using the Time Complexity Analysis method described in the methodology. This analysis leads to the identification of two critical modules from the algorithm : Matrix Multiplication and LU decomposition of a matrix. As directed by the methodology, software implementations of these algorithms are performed. These implementations are optimized using a number of techniques like Forward Store Optimization and Loop Unrolling. These optimized software models are compared with the hardware models built at the RTL Level of Abstraction. The main metrics of comparison are Execution Time, Power Dissipation and Time

to Market. After analyzing the results, a decision is made regarding the best way to implement the algorithm.

1.4 Thesis Organization

The thesis is organized in the following way:

Chapter 2 provides a description of the Design Space Exploration. It discusses the different hardware architectures with their important features. It also gives a brief idea about the various factors to be taken into consideration while choosing the right hardware architecture for a given application. Chapter 3 describes the design methodology for analyzing complex algorithms in detail. The reader is made familiar with the concept using lucid flow diagrams. Chapter 4 describes the case study of the FIR filter which shows comparison between tool generated and hand written C codes. It also describes the different kinds of optimizations that can be performed on the software description of a model to improve its performance. This case study forms the foundation of one of the important decision making points in the methodology described in Chapter 3. Chapter 5 deals with applying the methodology to the Attitude Determination and Control algorithm. The first part of the chapter gives a brief overview of the ADC algorithm. The next part describes the detailed analysis of the algorithm using the proposed methodology, and the final part gives a conclusion on the basis of the results obtained by analyzing the algorithm. Finally in Chapter 6, we conclude our work by giving an overview of our contribution and providing details of future work that can be done in this field.

Chapter 2

Design Space Exploration

The ever increasing complexity of signal processing and control algorithms has lead to the need for development of different methodologies to implement algorithms efficiently within a design cycle that is ever shrinking. The path from algorithm development to the actual implementation is a long process and needs careful decision making at various levels of abstraction in order to have a design which is an improvement over existing designs. The most important step in implementing an algorithm is choosing the right hardware platform onto which the algorithm can be mapped. This can be based on design specifications and requirements of the specific applications. The choice of the correct hardware architecture depends upon a number of important parameters, e.g., *Time to Market*, *Cost*, *Performance*, *Power Consumption* and *Feature Flexibility* [10]. After the choice of the correct hardware architecture, the next step is to select the correct methodology for design and implementation of the algorithm onto the selected hardware architecture. Currently, most designs are accomplished through the use of HDL-centric flows. However, device densities have increased at a pace that such flows have become both cumbersome and outdated. The need for a more innovative and higher-level design flow that directly incorporates model simulation with software/hardware implementation is needed [19]. This chapter focuses on the selection of the right hardware architecture for a specific ap-

plication based on several factors, and also on the different kind of high level design methodologies generally used to go from algorithm to design and implementation.

2.1 Choice of Hardware Architecture

There are numbers of factors which affect the selection of the correct hardware architecture for a specific application. For a space based application, these factors are *Time to Launch*, *Performance*, *Power Dissipation*, *Price* and *Feature Flexibility*. This section discusses the importance of each and every factor along with the tradeoffs between them.

2.1.1 Time to Launch

In the language of an aerospace engineer, the process of designing, fabricating and demonstrating the flight capabilities of a space based system is referred to as *Design Build and Fly*, abbreviated as DBF. The total time required to perform all these activities is referred to as Time to Launch. A significant amount of the total time is spent in the design process. The greatest threats to the field of aerospace engineering are technological obsolescence and ever intensifying global competition. In order to cope up with these threats, one of the main goals of the design team should be to shrink the design and development cycles.

2.1.2 Performance

Performance is an important criterion for space based applications, as the design should be able to meet the real time space constraints. Performance may be measured in many ways, but generically we look at millions of operations per second (MIPS), millions of multiply accumulates per second (MMACS), or, sometimes, the more simple millions of cycles per second (MHz) clock rate. Performance is lower on a

priority basis only when other key requirements such as low power or low cost increase in priority [10].

2.1.3 Price

Price includes the design costs and the Non Recurring Engineering (NRE) costs. A clear understanding of design costs can be indicated by giving an example of hardware and software designs. Software design involves describing an algorithm using some high level languages like C and C++. Hardware design involves describing the algorithms using hardware descriptive languages like VHDL and Verilog. The design has to be carried out such that the HDL description can be synthesized to produce the gate level netlist which can be mapped onto a hardware device. All this requires highly skilled engineers. The cost of the tools involved in the design process also adds to the total design cost. According to the International Technology Roadmap for Semiconductors (ITRS) 2003 edition, the NRE costs, including both design and manufacturing costs, in the deep submicron era, can easily amount to tens of millions of dollars [20] [21]. So, price is often the most obvious criteria of priority.

2.1.4 Power Dissipation

The most important requirement for any space based design is that it should have low power dissipation. Since the beginning of the 1990s, the reduction of power dissipation has become more and more important in chip design [22]. In addition to costly heat removal expense, excessive power consumption in systems also reduces the battery lifetime. As a result, the quality and reliability of a system would be severely compromised by high power dissipation. So, one of the main motives of all the design engineers is to develop new design techniques which can lead to the minimization of power dissipation in all types of applications. Not only the design technique, but also the architecture on which the design is implemented, plays a vital role in determining the overall power dissipation.

2.1.5 Feature Flexibility

Feature flexibility is the capability to modify or add features to meet changing requirements. Requirement changes occur more and more rapidly in today's markets. For example, early entry into standards-based products (such as communications or compression standards) before the standards are solidified is critical if manufacturers are to lead in the market. However, designers must build future proof products that can be upgraded to reflect the final standard after all nuances have been worked out. Thus you must have the feature flexibility required to update your product easily and quickly after market [10].

Feature flexibility is quite important for space based applications. A space satellite may be configured to perform a particular task, but it is possible that the task may have to be modified or replaced by another task in the future. The main factor to be taken into consideration for designing space based systems is that the hardware device for such systems should be reconfigurable and evolvable.

2.2 Architecture Exploration

The key requirements for an architecture to implement a complex signal processing or control algorithm are high performance, flexibility and energy efficiency. A recent Texas Instruments survey indicated that of the signal processing architectures considered by today's developer, the following alternatives were the most popular: ASIC, ASSP, configurable processor, DSP, FPGA, MCU and RISC/GPP [10]. Out of these architectures, the four most commonly used architectures are GPP, DSP, FPGA and ASIC. Programmable DSP architectures are flexible, but inefficient. Application Specific Integrated circuits (ASICs) are efficient, but inflexible. FPGAs make a tradeoff between ASIC and DSP by limiting their flexibility to a particular domain [20]. This section gives brief information about these architectures along with the advantages and disadvantages of using them for a particular application with respect to the various criteria described in the previous section.

2.2.1 General Purpose Processors (GPP/RISC)

General Purpose processors are programmable processors that can utilize software programmability to perform a wide variety of functions and hence have the greatest amount of flexibility. Their core architectures are not designed and optimized to perform a specific application. As a result of this, these processors are the least efficient of all the hardware architectures. They are judged fair in terms of Price and Power [10]. An example of a General Purpose Processor is an ARM Processor. An improvement to the class of general purpose processors is the development of Application Specific Instruction-set processors (ASIP) whose core's architecture and instruction set are customized in order to increase efficiency and reduce power dissipation [23].

2.2.2 Digital Signal Processors (DSP)

Digital Signal Processors are specialized microcomputers for real-time signal processing applications. Because of their specialized applications, programmable DSPs have evolved architectures that are significantly different from conventional microprocessors. On certain DSP benchmarks, their performance has consistently exceeded that of microprocessors with arithmetic co-processors by more than the order of magnitude throughout their ten year history [24].

A number of architectural innovations have been used to achieve this impressive performance. The most basic is the integration of fast multiplier/accumulator hardware into the data path; the arithmetic is not done on a processor which is separated from the main data path, but rather is an integral part of execution of every instruction. They have Harvard architecture with separate data and program buses with for the individual data and program memories respectively. They are referred to as first generation DSPs. The second generation DSPs retain much of the design of the first generation, but with added features such as pipelining, multiple arithmetic logic units (ALUs) and accumulators to enhance performance. TMS320C20 from Texas Instruments and Motorola DSP56002 are examples are second generation DSPs [25].

A further enhancement was observed with the evolution of the third generation DSPs. These DSPs have capabilities of executing Single Instruction Multiple Data (SIMD), Very Long Instruction Word (VLIW) and Superscalar operations. SIMD allows one instruction to be executed on many independent groups of data. For SIMD to be effective, programs and data sets must be tailored for data parallel processing, and SIMD is most effective with large blocks of data. VLIW processors issue a large number of instructions either as one large instruction or in a fixed instruction packet, and the scheduling of these instructions is performed by the compiler. Superscalar processors, on the other hand, can issue varying number of instructions per cycle and can be scheduled statically by the compiler or dynamically by the processor itself [25]. A further enhancement is observed in the latest TI TMS320C4x processors which combine both VLIW and SIMD into their architecture known as VelociTI.2. All these specialized architectures make DSPs faster and more power efficient than the General Purpose Processors.

2.2.3 Field Programmable Gate Arrays (FPGA)

FPGAs are integrated circuits that have undefined function at the time of manufacturing and must be programmed before it can be used. An FPGA is comprised of a two dimensional array of configurable logic blocks (CLB), which are connected by a configurable matrix of wire switches. Basic building blocks, such as adders and shifters, can be formed by configuring logic cells and interconnecting them. These basic building blocks can be used to form more complex designs.

The behavior of FPGA is specified using either a hardware description language or a schematic that is designed by Electronic Design Automation (EDA) tools. Design Automation tools are used to synthesize a HDL or schematic description to a netlist of logic cells for a target FPGA technology. Placement and Routing automation tools are used to allocate the logic cells in the FPGA device and the connections among these logic cells [20] [26]. FPGAs can achieve high performance and are quite efficient for algorithms that manipulate bits and have lots of parallelism. They are judged

good in terms of Time to Market and Feature Flexibility and fair in terms of Price and Power efficiency [10].

2.2.4 Application Specific Integrated Circuits (ASIC)

ASICs are manufactured on silicon wafers that can hold hundreds of silicon chips, which are called dies. They are developed using silicon compilers that can translate an HDL into a gate level netlist. ASICs can be made either from standard cell libraries or can be have full custom design. The full custom design will be more energy-efficient, but is very labor intensive [20].

ASICs represent the most efficient solution to implement a particular algorithm in hardware in terms of performance, area, and energy efficiency. The major drawback is their inflexibility. Once the ASIC is manufactured, its function cannot be changed. The Time to Market for an ASIC is high as compared to other architectures. A small modification requires developing a new ASIC.

Table 2.1 summarizes the characteristics of all the architectures with respect to the criteria discussed in Section 2.1

Table 2.1: Architectures and their Characteristics

	<i>Time to Market</i>	<i>Performance</i>	<i>Price</i>	<i>Power</i>	<i>Feature Flexibility</i>
GPP/RISC	Good	Good	Fair	Fair	Excellent
DSP	Excellent	Excellent	Good	Good	Excellent
FPGA	Good	Excellent	Fair	Fair	Good
ASIC	Poor	Excellent	Excellent	Good	Poor

The most important trade-off is between flexibility and efficiency as illustrated by Figure 2.1. Flexibility dictates design costs, time to market and reconfigurability, while efficiency determines performance and power dissipation. The figures of merit for measuring efficiency are MOPS/ mW (Energy Efficiency) and MOPS/ mm^2 (Area Efficiency), where MOPS is defined as Millions of Algorithmically defined Arithmetic Operations (e.g. multiply, add, shift) [2].

On one end of the spectrum are ASICs. They are specialized to a given application. Therefore, they have the least flexibility compared with other architectures. The lack of flexibility introduces high NRE costs, and a prolonged and expensive design flow. However, due to specialization, the efficiency is highest in terms of power and performance. On the other end of the spectrum are general purpose processors like Intel's Pentium 4. They can be organized and sequenced to implement arbitrary computations. This level of flexibility brings high-level programming language support, zero manufacturing and low design NRE cost for the application developer. However, they are too inefficient to deliver high performance and low power when applications have much parallelism [27]. Between these two extremes, architectures like DSP and FPGA try to make different compromises between efficiency and flexibility.

2.3 Implementation Techniques

This section discusses the three basic implementation techniques used commonly for mapping complex algorithms onto appropriate hardware architectures. Today's VLSI technology allows companies to build large, complex systems containing millions of transistors on a single chip. To exploit this technology, designers need sophisticated CAD tools that enable them to manage millions of transistors efficiently [4]. Until now, designers have been doing the design process at lower levels of abstraction like the transistor level and the Register Transfer Level (RTL). But, as the design complexity increases, it is not prudent to perform the design process at these levels as it reduces

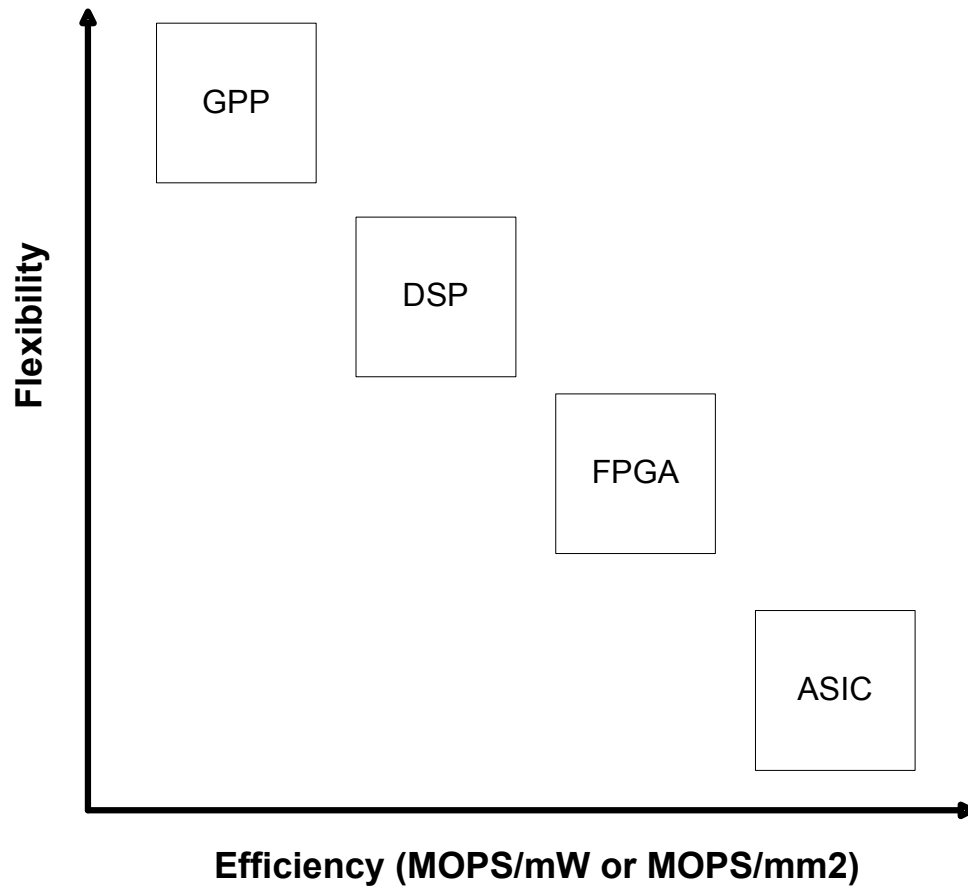


Figure 2.1: Flexibility vs. Efficiency Tradeoff [2]

the overall productivity of the design. The solution to this is to move up to the higher levels of abstraction. The advantage of this new methodology is that it allows us to design in a purely behavioral form, devoid of implementation details and then to map the design onto appropriate hardware architectures using the commercially available CAD tools.

In order to describe the design at the behavioral level, we need a tool which allows us to design and simulate the algorithm at the block level. Simulink is a software package for creating, editing and simulating dynamical systems on Matlab from *Mathworks*® [28] [29]. It is widely used in the field of aerospace engineering for system level designing. Simulink utilizes a graphical user interface (GUI) for building

models where models are created as block diagrams. It also allows complete models to be simulated by a variety of integration solvers. In addition, users can change parameters and immediately see what happens for “what if” exploration. While the simulation is running, users can see the result via display blocks, such as scopes. Based on the assumption that the high level description of the algorithm has been done in Simulink, the types of implementation can be divided into three main categories:

- Software Only Implementation
- Hardware Only Implementation
- Hardware-Software Co design

2.3.1 Software Only Implementation

In this type of implementation, the Simulink algorithm is converted into C code and compiled to produce the binary code which can be run on a Digital Signal Processor (DSP). The algorithm is converted to C code using the Real Time Workshop provided by Mathworks. Real-Time Workshop is a toolbox that is part of the Matlab Simulink package. It allows C code to be generated from block diagrams in Simulink. Once a system has been designed and simulated in Simulink, code for digital signal processors can be generated, compiled, linked, and downloaded to a DSP board [29] [30]. The process of generating C code from a Simulink model can be illustrated with the help of the flow diagram shown in Figure 2.2.

Real-Time Workshop converts the Simulink model stored in the .mdl file into .rtw file. This file contains all the necessary information about the Simulink model that is needed during code generation process. RTW also generates makefile (model.mk) from the system template makefile (system.tmf). After that, the Target Language Compiler (TLC) is invoked into the code generation process and it converts Simulink model to the C code. When the TLC concludes with the C code generation process, the *make* program, based on previously generated C files, *makefile*, and using the

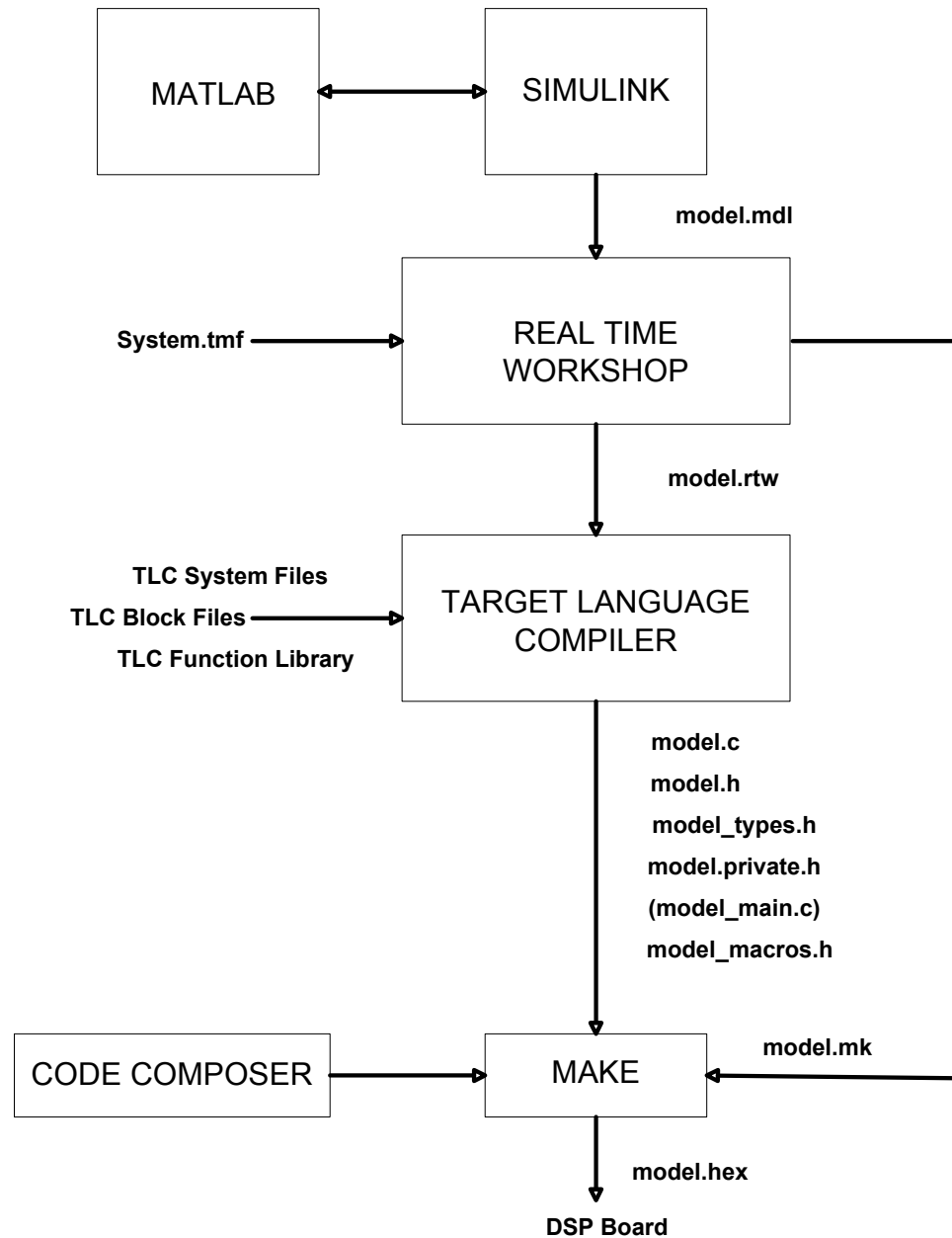


Figure 2.2: Code Generation Process from Simulink Model [3]

appropriate compiler, generates executable code. This code can be imported to a tool specific to the target DSP. An example of one such tool is the TI Code Composer Studio. The compiler present in this tool can be used to generate assembly code specific to the target DSP.

2.3.2 Hardware Only Implementation

In this type of implementation, the concept of high level synthesis is used. High-level synthesis is a sequence of tasks that transforms a behavioral representation into an RTL design. The design consists of functional units such as ALUs and multipliers, storage units such as memories and register files, and interconnection units such as multiplexers and buses. The block diagram in Figure 2.3 illustrates the flow of high level synthesis.

The compiler converts the behavioral description into an internal representation. The RTL library contains the physical and simulation models of components to be used during synthesis. The netlister generates the final RTL structure, consisting of a netlist of RTL components and a simulation model of each component. To verify the synthesized design's correctness, the designer can simulate the input description and the generated netlist by means of the simulation environment, an adjunct to the high-level synthesis system [4].

Xilinx System Generator and Simulink HDL coder are commercially available tools which facilitate the behavioral description and simulation of complex designs. Both the tools can be used to perform design at a level of abstraction higher than the RTL level. The design implementation using these tools is faithful in that the system model and hardware implementation are bit-identical and cycle-identical at sample times defined in Simulink. In System Generator, the capabilities of IP blocks have been extended transparently and automatically to fit gracefully into a system level framework. User-defined IP blocks can be incorporated into a System Generator model as black boxes which will be embedded by the tool into the HDL implementation of the design [31]. However, there exists no way to generate hardware from models which use the basic Simulink Blockset, and as such, greatly restricts the usefulness of these tools [19].

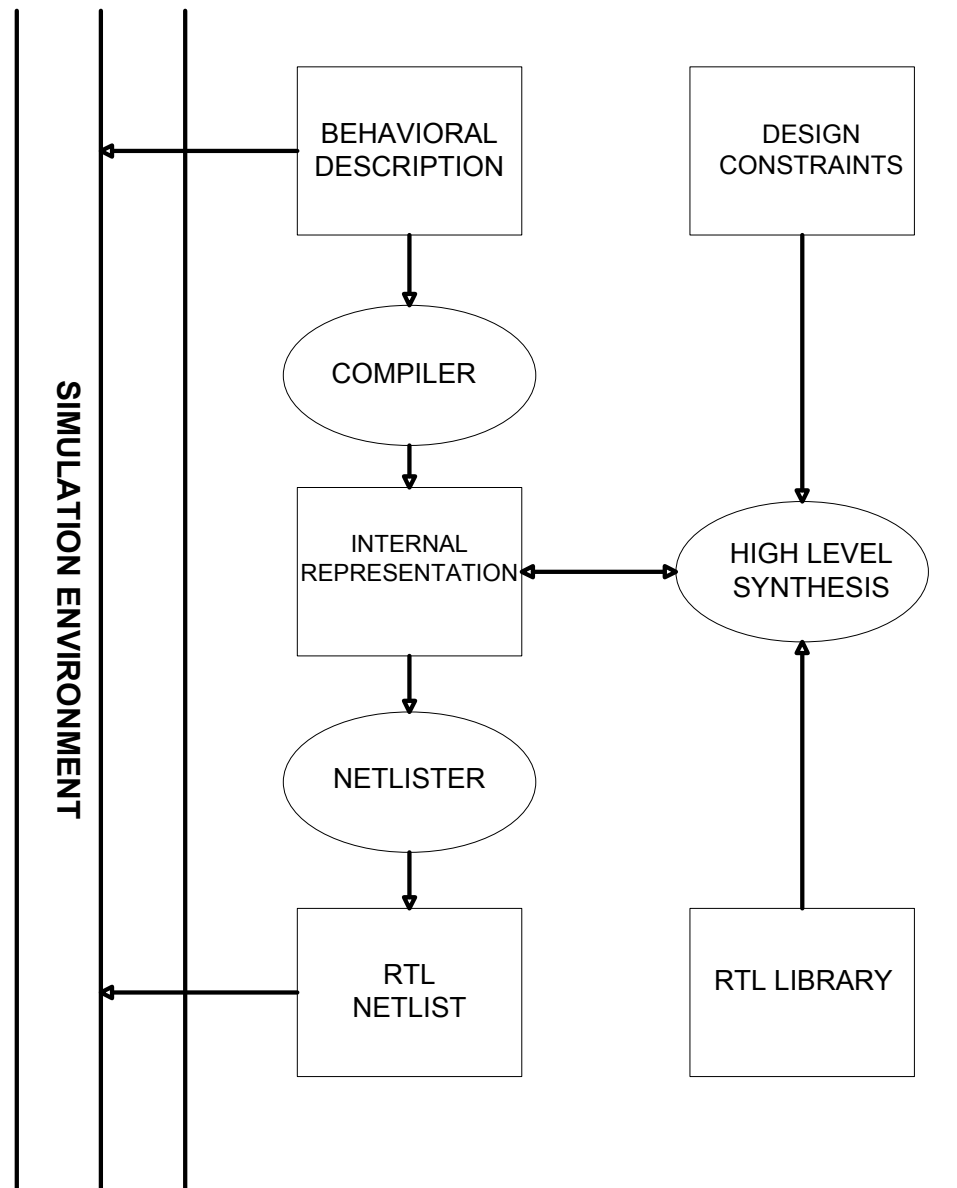


Figure 2.3: A generic high level synthesis system [4]

2.3.3 Hardware-Software Co-Design

Whenever we want to have a fast implementation of an algorithm, the best method is to have a software only implementation. The design time for this particular methodology will be very short. The disadvantage of this method is that it will not lead to an

efficient implementation of the algorithm. On the other hand, when Time to Market is not an important factor, one should go for Hardware only Implementation. The design time will be high but at the same time, the performance will be excellent. An intermediate between these two is the methodology of hardware-software codesign. It has advantages of both the methodologies.

Hardware-Software co-design, known as CoDesign methodology, takes into account the features of software and hardware elements in a unique design process. Some criteria are needed to split the system under development into hardware and software parts. The usual constraints are performances, hardware area or cost criteria. This process is usually called the “partitioning process”. After that, the software has to be compiled on the target architecture, and then the hardware has to be synthesized. A software and hardware evaluation is made to see if the performances and area requirements are reached. If not, the partition needs to be revised. The global scheme for hardware-software Codesign methodology is illustrated in Figure 2.4.

The first step in hardware-software partitioning is a profiling step to evaluate the relative computational complexities for the different blocks in the algorithm. Tasks which contribute to a large portion of the execution time of the processor are natural candidates for hardware implementation [32]. Profiling can be done using a number of profilers like the GNU gprof GCC profiler and the profiler in TI Code Composer Studio. The profilers can be used to give the execution times of the individual functions of the code. From the profiling results, the bottlenecks can be identified for implementation in hardware. In this manner, the performance of the overall system can be improved.

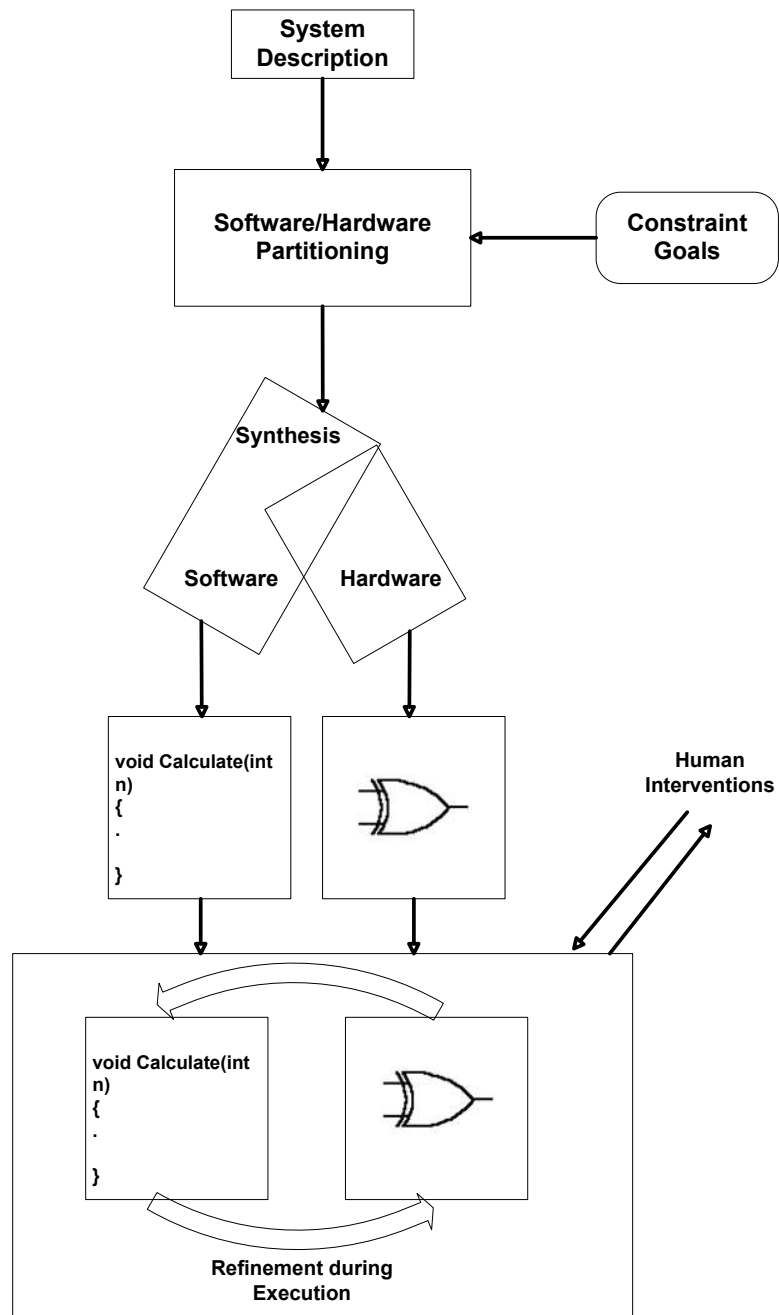


Figure 2.4: Global Scheme for Codesign Methodology

Chapter 3

Design Methodology

The main goal of this research is to develop a methodology that can assist a system engineer in implementing a complex algorithm on appropriate hardware architecture. The methodology described in this section is a step by step process which when applied to an algorithm, gives a clear understanding of the process of implementation of the algorithm. The validity of the methodology is shown by applying it to the Attitude Determination and Control [1] algorithm designed for the attitude control of small satellites.

Some of the architectures used for real time signal processing and control applications are General Purpose Processor (GPP), Digital signal Processor (DSP), Field Programmable Gate Array (FPGA), Application Specific Integrated Circuits (ASICs), Application Specific Instruction-set Processor and Reconfigurable Processor [10]. Out of these, the four most commonly used architectures for majority of applications are GPP, DSP, FPGA and ASIC. The algorithm can be implemented on any of these architectures. This methodology shows the process of analyzing the algorithm and deciding the platform(s) on which the algorithm can be implemented.

The methodology described here assumes that the application model has been

designed and simulated using Simulink from MathWorks. The entire flow is divided into six main steps. The steps described in the methodology are specific to TI DSPs and Xilinx FPGAs. The reasons for choosing TI and Xilinx devices are listed below:

- Both TI and Xilinx provide state-of-the-art technology devices. The latest TI DSP TMS320C67xx series has the VelociTi architecture which is a combination of both VLIW (Very Long Instruction Word) and SIMD (Single Instruction Multiple Data) architectures. The Virtex-4sx FPGA from Xilinx is best suited for DSP and low power applications.
- There are robust tools associated with these devices which enable fast and efficient implementations of designs on these device architectures.
- It is assumed that the algorithms are designed and simulated using Simulink from Mathworks. Both TI and Xilinx have tools which are compatible with the toolboxes present in the Matlab Simulink package.

Each and every step of the methodology will be explained in detail with use of examples to enhance the understanding of the process. The methodology assumes that the design engineer has a good knowledge of software (C) and hardware descriptive (Verilog / VHDL) languages. A number of steps in the process involve the use of some commercially available tools like Xilinx ISE [33] and TI Code Composer Studio [34]. The engineer is not expected to know the functionality of the tools. A brief overview of the tools will be provided when they are referred to in the methodology.

The flow described involves a comparison of hardware and software implementations. The comparison is on the basis of three important factors: Execution Time (Speed), Power Dissipation, and Development Time (a part of the total Time to Launch for space based applications). Another factor which is of significance for space based applications is the size (area) of the design. It is also touched upon in the methodology. We feel this methodology is of value because a novice engineer can follow the steps described in this methodology and implement the given algorithm.

Figure 3.1 shows the major steps of our design methodology. A detailed description of all the steps is given in the section below.

3.1 Proposed Methodology

Step 1: Selection of the candidate platforms (architectures)

The first step in the methodology is to narrow the selection of platforms that can be chosen as candidates for algorithm implementation. The Simulink model of the algorithm at this stage is a general description of the algorithm irrespective of the platform on which it can be implemented; hence it is called Platform Independent Model (PIM). We present a simple technique which helps to select the right architecture(s) based on the requirements of the application. This technique is based on data provided by Texas Instruments. The choice of appropriate architecture(s) depends upon criteria like Performance, Power Dissipation, Cost, Time to Market and Feature Flexibility.

This methodology focusses on algorithms for space based applications in which the most important criterion is low power dissipation. Other criteria like Performance and Development Time are also fairly important. Let us consider that weights are given to these criteria on the basis of requirements of the application. The weights are between 0 and 1. 0 means no importance and 1 means most important. We take four criteria into consideration: Performance, Power Dissipation, Development Time and Flexibility. Let the weights given to these criteria be the inputs of a system which selects the candidate platforms. The system consists of a matrix of numbers, which are ratings given to the four architectures, obtained on the basis of Table 3.1. The numbers are relative to each other.

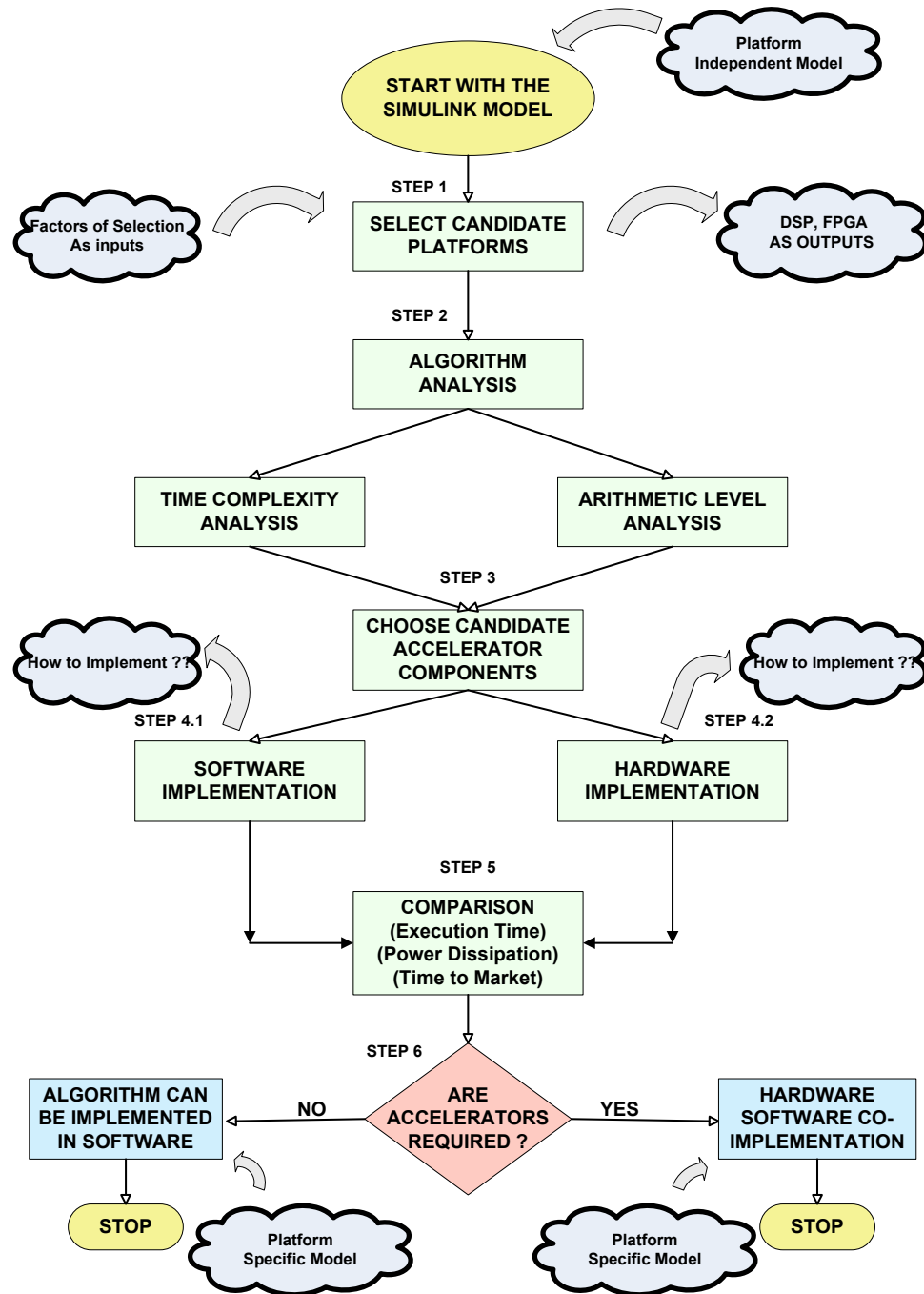


Figure 3.1: Overall flow diagram of the Proposed Methodology

The input given to the system is a 4×1 matrix as shown below

$$\begin{pmatrix} \text{Weight given to Time to market} = 0.6 \\ \text{Weight given to Performance} = 0.8 \\ \text{Weight given to Power consumption} = 1.0 \\ \text{Weight given to Flexibility} = 0.4 \end{pmatrix}$$

Table 3.1: Characteristics and their numerical values

<i>Characteristics</i>	<i>Numerical Value</i>
Excellent	1
Very Good	0.8
Good	0.6
Fair	0.4
Poor	0.2

Table 3.2: Tabular representation of the system matrix derived from data from TI and Table 3.1

	<i>Time to Market</i>	<i>Performance</i>	<i>Power</i>	<i>Feature Flexibility</i>
GPP	0.6	0.6	0.4	1.0
DSP	1.0	0.8	0.6	1.0
FPGA	0.6	1.0	0.4	0.6
ASIC	0.2	1.0	0.6	0.2

We use a technique, called *Concept Scoring* [35], to select the best architecture for the purpose of implementation. In this technique, the input weights are multiplied by the ratings given to each architecture to obtain weighted scores for all the criteria. The total score for each architecture is obtained by adding the individual weighted scores for all the criteria. This can be seen in Figure 3.2. Here W.Score refers to the weighted scores obtained by multiplying the input weights with the ratings given to each architecture.

The total scores obtained in this example are 1.64, 2.24, 1.8 and 1.6. These are

Selection Criteria	Weight	GPP		DSP		FPGA		ASIC	
		Rating	W.Score	Rating	W.Score	Rating	W.Score	Rating	W.Score
Time to Market	0.6	0.6	0.36	1.0	0.6	0.6	0.36	0.2	0.12
Performance	0.8	0.6	0.48	0.8	0.64	1.0	0.8	1.0	0.8
Power Dissipation	1.0	0.4	0.4	0.6	0.6	0.4	0.4	0.6	0.6
Feature Flexibility	0.4	1.0	0.4	1.0	0.4	0.6	0.24	0.2	0.08
Total Score		1.64		2.24		1.8		1.6	

Figure 3.2: Concept Scoring Matrix

the scores for the four architectures GPP, DSP, FPGA and ASIC in that order. It can be seen that DSP has the highest score of all. This indicates that DSP is the ideal candidate for implementing the algorithm. But as far as performance is concerned, FPGA has higher performance than DSP. Also, it has the second highest weighted score as observed from the data obtained. So, a clear decision cannot be taken as to which is the right architecture onto which the algorithm can be mapped. The solution is to implement the computationally complex parts of the algorithm on FPGA and the rest of the algorithm on DSP. The details regarding this aspect are discussed in the ensuing steps of the methodology.

The advantage of this technique is that it narrows down the window of selection of the target architectures. GPP and ASIC have low weighted scores, so they are eliminated from the list of candidate architectures.

Step 2: Algorithm Analysis

The second step in the methodology is to perform the analysis of the Simulink model provided. In the first step, it was deduced that DSP and FPGA are the candidate architectures on which the algorithm can be implemented. Since the performance of FPGA (as seen in Table 3.2) is better than that of the DSP, it can be used to implement the complex parts of the algorithm, and DSP can be used to implement the rest of the algorithm. In order to determine the computationally complex parts of the algorithm, the algorithm has to be analyzed.

To analyze an algorithm is to determine the amount of resources (such as time and storage) necessary to execute it. Most algorithms are designed to work with inputs of arbitrary length. Usually the efficiency or complexity of an algorithm is stated as a function relating the input length to the number of steps or storage locations. Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm for solving a given computational problem. These estimates provide an insight into directions for efficient implementation of algorithms.

The process of complexity analysis of an algorithm can be illustrated with the help of the flow diagram in Figure 3.3. It has been deduced in Step 1 that DSP and FPGA are the candidates for implementing the algorithm. The following steps give a brief overview of the complexity analysis process illustrated by the flowchart in Figure 3.3.

1. Use Real Time Workshop from Mathworks to generate C code from the Simulink model. The process has been described in Section 2.3.1.
2. The next step is to import this C code into TI Code Composer. Select the target Digital Signal Processor for which the code needs to be compiled. This is done by including the target library in the Code Composer Project.

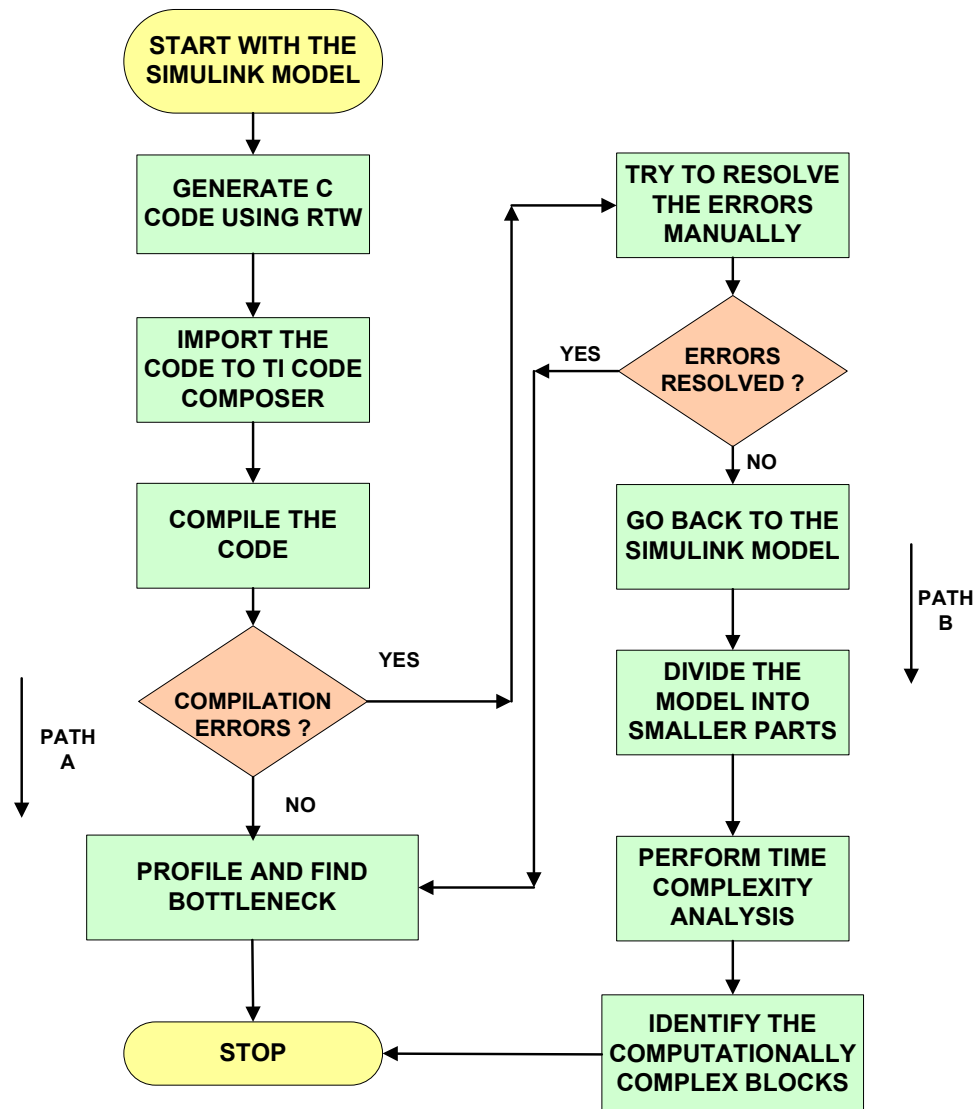


Figure 3.3: Algorithm Analysis Flow

code. These data structures are not recognized by the built-in compiler of Code Composer Studio.

5. If there are no compilation errors, the next step is to profile the C code. This is denoted by Path A in the flow diagram. The Code Composer Studio Profiler analyzes the execution of the C program and shows where the “hotspots”, or the areas where a large portion of the cycles being used, occur. The profiler can show the developer how many cycles a function takes to execute, as well as how often it is called.
6. If there are compilation errors, the next step is to resolve the errors. This depends upon the number and types of errors. In order to resolve the errors, it is required to understand the C code generated by RTW.
7. If the code generated by RTW is difficult to understand, and the errors cannot be resolved, the next step is to analyze the algorithm using the divide and conquer method. This is denoted by Path B in the flow diagram. Divide and conquer is a technique for analyzing algorithms that consists of dividing the problem into smaller sub-problems which are identical to the original problem but smaller in size, and then analyzing the sub-problems independently of each other.
8. If Path B is chosen, the last step is to find out the computationally complex blocks from the model using Time Complexity Analysis. An overview of Time Complexity Analysis is presented below.

Time Complexity can be defined as a measure of the amount of time required to execute an algorithm. It gives an approximate estimate of the number of operations needed to complete the computation of the given algorithm. It is independent of the programming language chosen to implement the algorithm, the quality of the compiler, and the speed of the computer on which the algorithm is to be executed. Formally, the Time Complexity $T(n)$ of an algorithm is $O(f(n))$ (of the order of $f(n)$) if, for some constant C and for all but finitely many values of n , $T(n) \leq C * f(n)$.

This gives an upper bound on the amount of work done for sufficiently large n . The following examples illustrate the method of determining the Time Complexity of an algorithm.

Example 1: Matrix Addition

Consider the addition of two matrices A and B of m rows and n columns using the following algorithm.

Procedure: MATADD (A, B: matrix; var C: matrix);

Begin

for $i=1:m$ *do*

for $j=1:n$ *do*

$C[i,j] = A[i,j] + B[i,j];$

End

In this algorithm, we need

- $m \times n$ additions
- $2 \times m \times n$ memory reads
- $m \times n$ memory writes

Thus the time complexity of the procedure matrix addition is

$$\begin{aligned}
 & (t_1 \times m \times n) + (t_2 \times 2 \times m \times n) + (t_3 \times m \times n) \\
 &= (m \times n) (t_1 + 2t_2 + t_3) \\
 &= O(m \times n)
 \end{aligned}$$

where,

t_1 is the time for one addition

t_2 is the time for one memory read,

t_3 is the time for one memory write

Example 2: Polynomial Evaluation

Consider the polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \dots + a_2 x^2 + a_1 x + a_0$$

where a_n is non-zero for all $n \geq 0$

The first term in the polynomial requires n multiplications. The second term requires $(n-1)$ multiplications. The third term requires $(n-2)$ multiplications. Likewise, the third last term requires 2 multiplications and the second last term requires a single multiplication. The total number of multiplications required is given as

$$\begin{aligned} T(n) &= (n) + (n-1) + (n-2) + \dots + 2 + 1 \\ &= n(n+1)/2 \\ &= n^2/2 + n/2 \end{aligned}$$

This is an exact formula for the maximum number of multiplications. We say that the number of multiplications is on the order of n^2 , or $O(n^2)$. The following table shows the different categories of Time Complexities.

Step 3: Choose Candidate Accelerator Components

As illustrated in the flow diagram of Figure 3.3, there are two different paths (A or B) which can be taken to perform the complexity analysis of the algorithm. This step of the methodology discusses about selection of the candidate accelerator components from the results of the complexity analysis. This section is divided into two parts. Part A describes the method of selecting the accelerator components when Path A in Figure 3.3 is taken, and Part B describes the method of selecting the accelerator components when Path B in Figure 3.3 is taken.

Table 3.3: Different Categories of Time Complexities

$O()$	<i>Category</i>
$O(1)$	Constant Time Algorithm
$O(\log N)$	Logarithmic Time Algorithm
$O(N)$	Linear Time Algorithm
$O(N \log N)$	Linearithmic Time Algorithm
$O(N^a)$ $a > 1$ and a is an integer	Polynomial Time Algorithm
$O(b^N)$ $b > 1$ and b is a real number	Exponential Time Algorithm

Part A

This part discusses the method of selecting the candidate accelerator components using the profiling results generated by the Code Composer Studio profiler. Let us consider that the C code of the model consists of six functions. The profiler gives information in terms of the number of target processor clock cycles required by all the functions to process certain amount of input data. Suppose the functions in the C code of the model are f1 through f6, and the number of processor clock cycles required to process 10 samples of input data by each function are 10, 200, 5000, 250, 300 and 1000 in that order, then, it can be seen that the function f3 consumes maximum number of clock cycles. In other words, it is the "hotspot" of the C code. The conclusion is that this function is the candidate accelerator component and can be implemented in hardware on FPGA and all the other functions can be implemented in software on DSP. This is shown in Figure 3.4. The arrows indicate the flow of data between DSP and FPGA.

Part B

This part discusses the method of selecting the candidate hardware accelerator components using Time Complexity Analysis. Path B in Figure 3.3 talks about

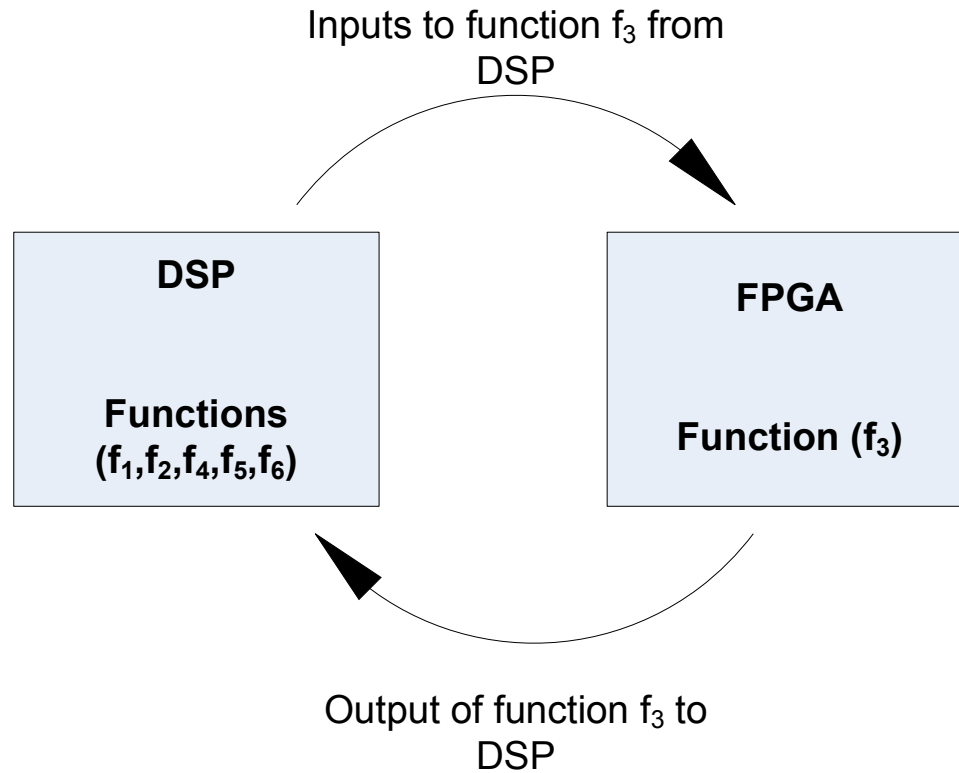


Figure 3.4: Flow of data between DSP and FPGA

dividing the Simulink model into small parts and then analyzing each one of them individually. Each and every Simulink block in the small model can be treated as an algorithm by itself. Consider the model shown in Figure 3.5.

It has adders, multiplexers, matrix concatenation blocks and a matrix addition block. Each and every block in this model can be considered as an separate algorithm and its time complexity can be determined individually. The time complexity of an adder is $O(n)$. Matrix concatenation is a constant time block as it simply involves concatenation of elements which does not involve any mathematical operations. The most critical block in this model is the matrix addition block. This block adds two 2×2 matrices. This block has a time complexity of $O(n^2)$ (as shown in the example in the previous section where $m=n$) where $n = 2$ in this case. Hence the block which consumes maximum amount of computation time in this model is that of matrix

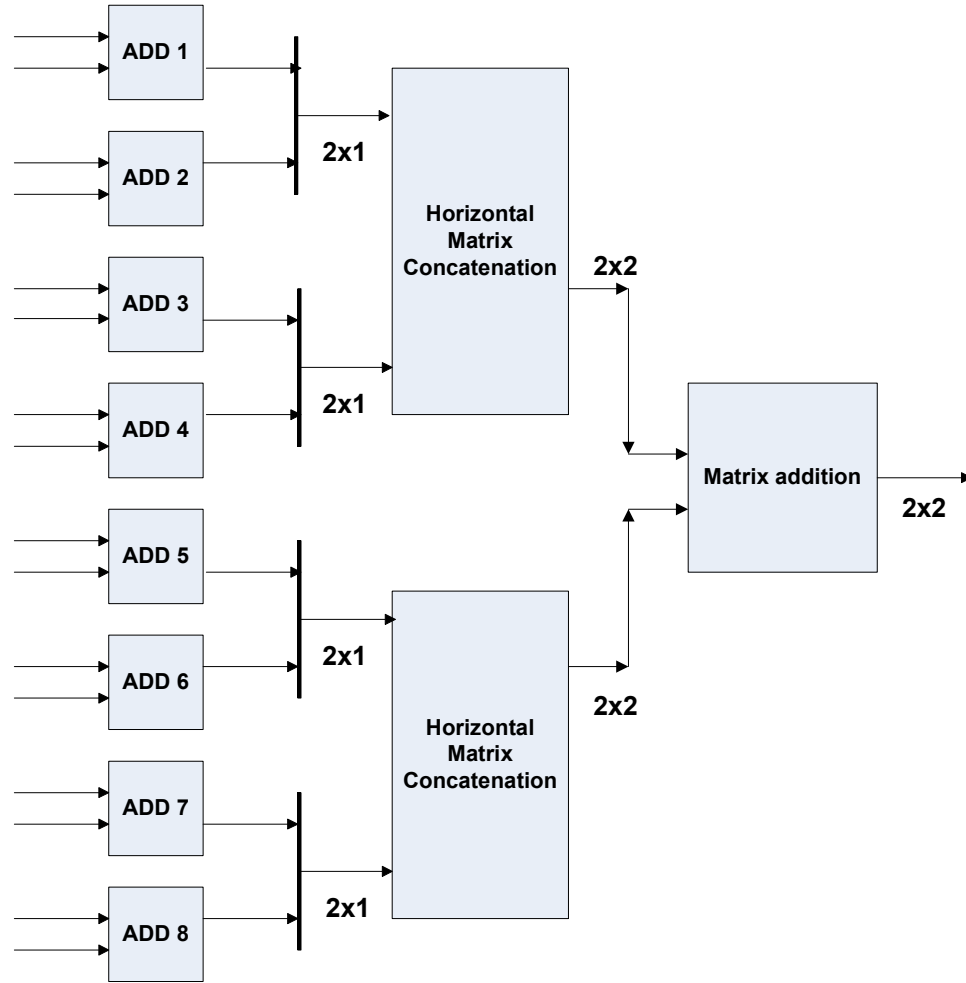


Figure 3.5: A simple Simulink model to explain the concept of Time Complexity Analysis

addition. The following set of rules is used to compare the complexities of the various blocks.

- n dominates $\log_2(n)$
- $n \log_2(n)$ dominates n
- n^2 dominates $n \log_2(n)$
- n^m dominates n^k when $m > k$
- a^n dominates n^m for any $a > 1$ and $m \geq 0$

That is, $\log_2(n) < n < n\log_2(n) < n^2 < \dots < n^m < a^n$ for $a \geq 1$ and $m > 2$

In this manner, the blocks with maximum time complexity are chosen as the candidate accelerator components. This operation is performed on all the small models obtained by dividing the large Simulink model.

Step 4: Perform Software\Hardware Implementations of Accelerator Components

This step involves performing software and hardware implementations of the accelerator components selected in Step 3. Both the software and hardware implementations can be done in two different ways. This step has been divided into two parts. Step 4.1 gives details of the software implementation and Step 4.2 gives details of the hardware implementation.

Step 4.1: Software Implementation

The basic aim of performing the software implementation of the accelerator components is to compare its performance with its hardware implementation. The overall process of software implementation of the candidate accelerator components is illustrated in the Figure 3.6. What follows is a brief overview of all the blocks.

The software implementation of the accelerator components can be done in two different ways. This choice depends upon the path taken by the designer in Step 2. If path A is taken (which identifies the accelerators on the basis of the Code Composer profiling results), the designer should choose option 1 in Figure 3.6. On the other hand, if path B is taken in step 2 (which identifies the accelerators using Time Complexity Analysis), the designer should choose option 2 in Figure 3.6. A detailed analysis of the comparison between option 1 and option 2 in terms of performance of the C code will be shown in Chapter 4. After generating C code for the accelerator component, the next step is to compile it and profile it using the Code Composer

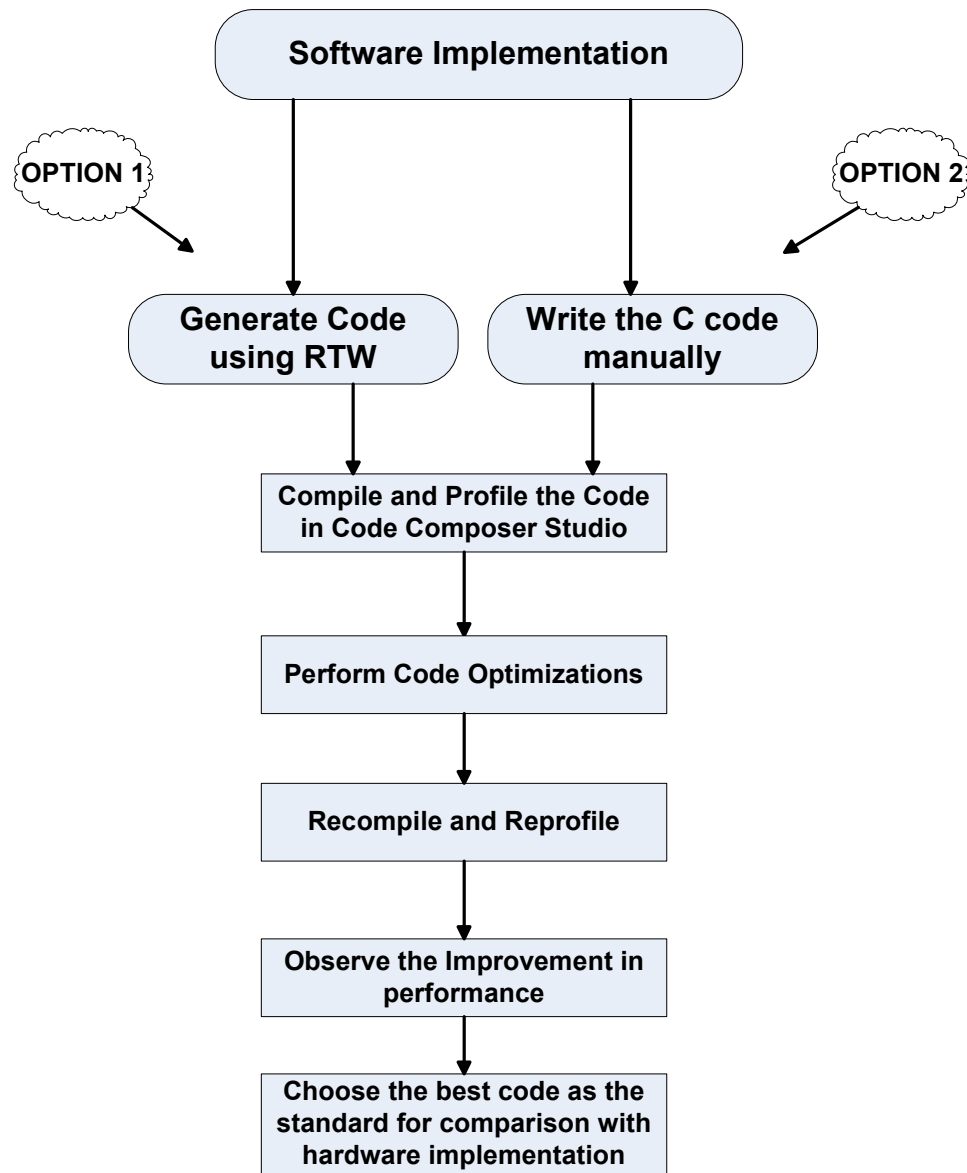


Figure 3.6: Software Implementation Process

Studio profiler. Then, the number of processor clock cycles required to execute the code for particular input data is observed. The next step is to optimize the C code to improve its performance.

Optimization of the code involves use of techniques that can reduce the number

of clock cycles required to execute the code. One of the disadvantages of C compilers in taking full advantage of high-performance DSP architectures is that they cannot always efficiently map complex algorithms to the multiple VLIW functional units. In order to overcome this limitation of the compilers, the designer has to modify the C code in such a manner that the resulting assembly code generated by the compiler can take maximum advantage of the underlying architecture. This can be done by performing different optimizations on the C code like Loop Unrolling [36], Forward Store Optimization [37] and use of intrinsic functions. These techniques will be explained in detail in Chapter 4 with the help of a case study.

After performing optimizations on the C code, it is again compiled and profiled to observe the improvement in performance. This process is repeated several times till we reach the point where performing any optimization does not improve the performance of the code any more. The optimized code which takes the least number of processor clock cycles to process a particular set of input data is chosen as the standard for comparison with hardware implementation.

Step 4.2: Hardware Implementation

After performing the software implementation of the accelerators, the next step is to do their hardware implementation on an FPGA. The hardware implementation process takes more time than the software implementation process and requires careful consideration at each step in order to have an efficient design. The flow diagram in Figure 3.7 illustrates the main steps in the hardware implementation of a design. This flow can be used to perform implementation of designs on Xilinx FPGAs.

Like the software implementation, the hardware implementation of the accelerator components can be done in two different ways. The first option is to use Xilinx System Generator [31] to perform block level design of the accelerator. A brief overview of Xilinx System Generator has been given in Chapter 2. The advantage of using System Generator is that it leads to faster implementation of the design. But it has certain

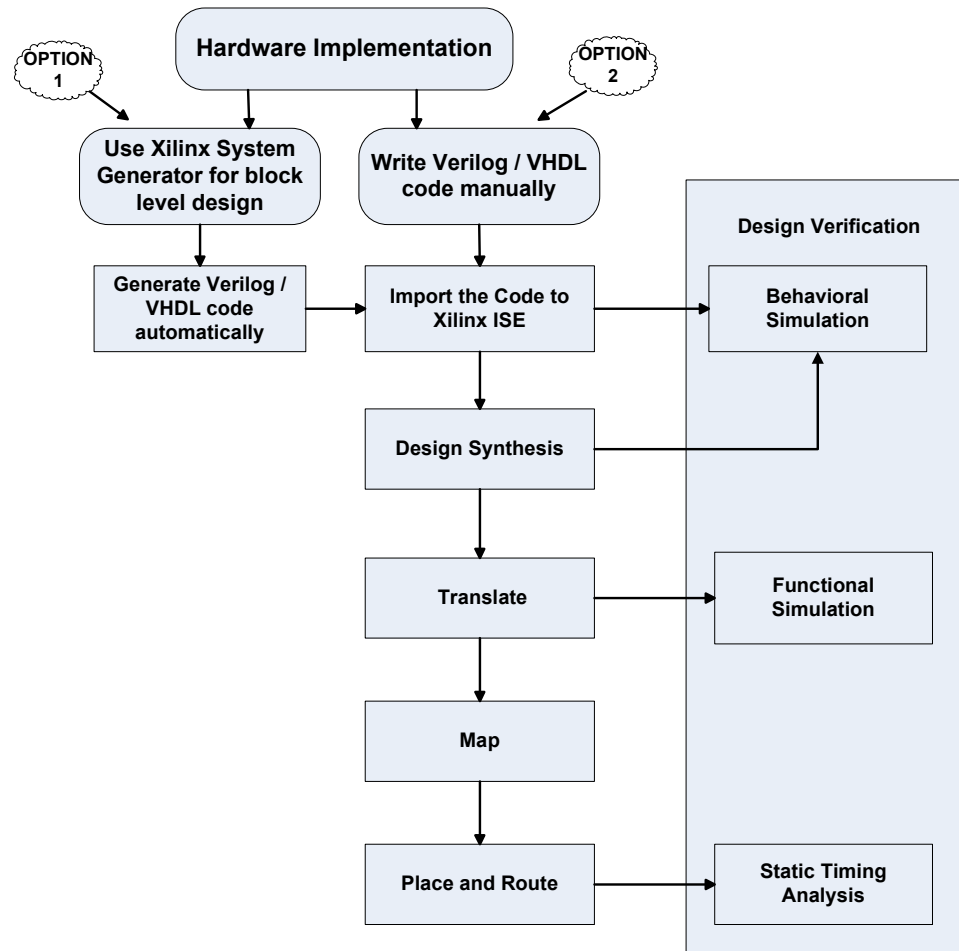


Figure 3.7: Hardware Implementation Process

disadvantages. One might view System Generator as a high-level design tool in which high-level function blocks are connected together while the tool handles the remaining details. This view is inconsistent with our experience. While the design entry method is high-level, the designs themselves typically are not. First of all, it does not have blocks corresponding to all the Simulink blocks in its libraries. Secondly, many of the existing high level blocks of System Generator are too constrained and cannot be parameterized. As an example, the existing FFT block provided with System Generator can only handle 16-bit input and 16-bit output, and requires a continuous stream of input data without a data valid indicator. This causes the designers to build

designs entirely from the basic building blocks, blocks similar to the functionality already available in HDLs. In addition, the amount of control logic necessary to handle delays and clocking often requires low-level manipulation by the user. The second option is to perform hardware implementation of the accelerator components using hardware descriptive languages, e.g. Verilog and VHDL. It consumes more time than the first option, but leads to an efficient implementation of the design.

The second step in the process shown in Figure 3.6 is to import the automatically generated (option 1) or manually written (option 2) Verilog / VHDL code to Xilinx ISE [38] and perform behavioral simulation of the code. This first pass simulation is typically performed to verify RTL (behavioral) code and to confirm that the design is functioning as intended. The next step is to perform synthesis of the design. The synthesis process checks code syntax and analyzes the hierarchy of the design which ensures that the design is optimized for the design architecture selected. The resulting netlist is saved to a .ngc file (A file that contains both logical design data and constraints). The .ngc file is then converted to an .ngd file. The .ngd file is essentially a netlist of primitive gates, which could be implemented on any one of a number of types of FPGA devices Xilinx manufactures. The next step is to map the primitives onto the types of resources (logic cells, I/O cells, etc.) available in the specific FPGA being targeted. The output of the Xilinx map tool is an .ncd file. The design is then placed and routed, meaning that the resources described in the .ncd file are then assigned specific locations on the FPGA, and the connections between the resources are mapped into the FPGAs interconnect network. The delays associated with interconnect on a large FPGA can be quite significant, so the place and route process has a large impact on the speed of the design. The output of the place and route engine is an updated .ncd file, which contains all the information necessary to implement the design on the chosen FPGA. This completes the process of hardware implementation of the design on FPGA.

An important factor to be taken into consideration while designing the hardware is deciding whether to implement the hardware using fixed-point or floating-point

arithmetic. Unless very large dynamic range is required, the implementation should be done using fixed-point. Fixed point implementations have higher clock rates, less silicon area usage, simpler functional units, cheaper implementation and low power consumption when compared to their floating point counterparts [39]. In fixed-point representation, proper choice of format using which data is represented is very important. Usually fixed-point fractions are denoted using the Q-format where Q denotes Quantity of Fractional bits [40]. Normally, Qm.n indicates m bits for the integer while n denotes the number of bits devoted to the fractional part. The choice of format depends upon the type of inputs and the type of operations to be performed in the design. Let us consider an example of a 32 bit fixed point number.

- If inputs to the design are normalized (between -1 and 1), the most appropriate data format would be Q0.31. In this format, the largest positive value is 0.999 and the least negative value is -1.
- But, if the division operation needs to be performed in the system, then choice of this format is not appropriate. This can be shown by the following example.
 Dividend = 0.5 which in Q0.31 format is 0.100000000000000000000000000000
 Divisor = 0.25 which in Q0.31 format is 0.010000000000000000000000000000
 The result of this operation should be 2.0. But, the largest value that can be represented using this data format is 0.999. Hence, use of this data format can lead to overflow.
- The solution to this problem is choosing another data format, let us say, Q16.16. The same example is shown using this data format.
 Dividend = 0.5 which in Q16.16 format is 0000000000000000.1000000000000000
 Divisor = 0.25 which in Q0.31 format is 0000000000000000.0100000000000000
 The result can be represented very well using this data format.
 Result = 0000000000000010.0000000000000000

Step 5: Comparative Metrics measurement

The next step is to measure metrics like execution time, power dissipation and time to market for both hardware and software implementations to have a comparison between both types of implementations on the basis of these metrics. Execution time for both types of implementations can be measured using the formula shown below.

$$\text{Execution time} = \frac{\text{Number of simulation clock cycles}}{\text{Maximum clock frequency}}$$

Time to Market is measured in terms of the amount of development time required for both types of implementations. The most critical metric of comparison for our comparison is power dissipation. The measurement of power dissipation for both types of implementations is a tricky process and requires a number of assumptions to be taken in order to have a commensurate comparison. The measurement of these metrics for both software (DSP) and hardware (FPGA) implementations is discussed below.

A. Execution Time

Execution time of a computational task is the amount of time a task executes on a specific hardware. The method of calculation of execution time for both software and hardware implementations is illustrated in the following steps:

Software Implementation (DSP)

- Profile the C code of the accelerator components.
- Observe the number of clock cycles it takes to process the input data.
- Divide the number of clock cycles by the maximum operating frequency of the DSP.

Example: A function f takes 4000 clock cycles to process 20 samples of input data. If the processor on which the function is executed operates at a maximum frequency

of 225 MHz, then the execution time can be given as

$$\text{Execution Time} = 40 / 225 \text{ MHz} = 177.78 \mu\text{s}.$$

Hardware Implementation (FPGA)

- Observe the number of clock cycles required to perform a given operation using Functional Simulation.
- Implement the design on FPGA (which involves Translate, Map and PAR)
- Observe the maximum frequency at which the design operates in the Static Timing Report.
- Divide the number of clock cycles by the maximum frequency at which the design operates.

Example: If an operation takes 30 clock cycles to perform a particular task, and the design works at a maximum frequency of 100 MHz, then the execution time is given as $\text{Execution Time} = 30 / 100 \text{ MHz} = 0.33 \mu\text{s}$.

B. Power Dissipation

Low Power Dissipation is one of the crucial requirements for any space based application. There are specialized tools available for measuring power dissipation for both DSP and FPGA. The technique of power dissipation measurement for both software (DSP) and hardware (FPGA) implementations is discussed below.

DSP Power Consumption

The power dissipation in the case of software implementation depends upon the underlying DSP architecture on which the code is executed. The total power consumption for a given application consists of two main components:

- **Baseline Power:** Baseline consumption is power that is consumed that is not dependent on chip activity. This includes things like static power, Phase Locked Loop power, and clock tree power. While independent of activity, baseline power

is dependent on voltage and temperature. Thus, the user can determine base-line power only by the selected part (and CVDD voltage) or by the operating temperature [41].

- **Activity Power:** Activity consumption is power that is consumed by active parts of the DSP (the CPU, EDMA, peripherals, etc.). Activity power is independent of temperature, but dependent on voltage and switching levels. This power consumption is dependent upon the percentage utilization of the different parts of the DSP for running a particular code [41].

TI provides power estimation spreadsheet for estimating power consumption for a particular application. The inputs to the power estimation spreadsheet are:

- “Frequency” - The operating frequency of a module or the frequency of external interface to that module.
- “Utilization” - The relative amount of time the module is active or in use versus off or idled.
- “Write” - The relative amount of time (considering active time only) the module is sending data out of the DSP versus reading data into the DSP.
- “Bits” - The relative amount of time (considering active time only) the module is sending data out of the DSP versus reading data into the DSP.
- “Switch” - The probability that any one data bit will change state from one cycle to the next.

In order to have a commensurate comparison of power consumption between FPGA and DSP, it will be assumed that the power consumed in DSP will be due to activity level changes in the CPU core and not any other part of the DSP. This means that the power consumption in any other module of the DSP will be ignored. A significant amount of power is dissipated by the peripheral components of the DSP while reading from or writing data to external memories. In the case of FPGA

implementation, the design has been implemented such that there is no transfer of data between the FPGA fabric and external memories. This leads us to ignore the power dissipation in the peripheral components of the DSP in order to have a fair comparison between both types of implementations.

The CPU utilization isn't as straightforward, because there are varying degrees of use for the CPU. Here, 0 percent utilization means the CPU is idle and does no useful work, where 100 percent utilization means all 8 functional units are active every cycle and the maximum amount of data is brought in from cache memory every cycle. Few DSP algorithms will achieve 100 percent utilization, because this requires everything to be used every cycle, with no stalls. Even DSP-intense applications do not spend all of the time in such highly parallel loops. Time is typically also spent executing control code or less demanding DSP algorithms. These types of code may execute only a few instructions in parallel and significantly reduce the I/O of the CPU, and thus reduce overall utilization. Therefore, the balance of CPU use for the application must be considered, and entering 100 percent utilization is not practical for real applications. For most of the applications, the percent utilization is measured by multiplying the average percent utilization (which is 60 percent in most cases) with the percent of time spent in running the algorithm to process input data [41]. This can be explained with the help of the following example:

Consider that the total number of clock cycles required to run a program on a DSP is 4000. Out of these 4000 clock cycles, 2000 clock cycles are utilized by a function which calculates the inverse of a 3×3 matrix. So the percent of time spent in running the matrix inverse algorithm is 50 percent. This is multiplied with the average percent utilization of the CPU core (which is 60 percent). So the value to be entered in the power estimation spreadsheet is $0.5 \times 0.6 = 0.3$ which is 30 percent. The power estimation spreadsheet gives the total power consumption as a split of the baseline power and the activity power.

FPGA Power Consumption

The estimate of power consumption of a Xilinx FPGA design can be done using a tool called XPower [42] from Xilinx. XPower calculates power as a summation of the power consumed by each element in the design. The power consumed by each switching element in the design is given by:

$$P = C * V^2 * E * F * 1000$$

where:

P=Power in mW

C=Capacitance in Farads

V=Volts

E=switching activity (average number of transitions per clock cycle)

F=Frequency in Hz

Capacitance is determined by the user's design (and for a design implemented in a specific device, fixed during characterization of the routing resources and the elements in the device). Voltage is a fixed value for a specific device set by default in the XPower interface. $F * E$ is the total number of transitions for a specific element. Frequency, or the activity rate of each signal in a design, is the most variable element of the above equation for this tool. XPower determines the switching activity of elements in the design by using any of the following data provided by a user:

- Global default activity rate
- Simulation results generated through a .VCD file
- Activity rates manually entered through the GUI

Where "activity rate" is the rate at which a net or logic element switches. Figure 3.8 shows the inputs required by XPower to calculate the power consumption of a design. The abbreviations used in the figure are explained below.

- **NCD**: FPGA design file that provides the design topology and physical resource usage. This file is generated during Implementation.

- **PCF**: Design file that provides clock frequency and voltage. This file is generated during implementation.
- **.VCD**: Design file that provides detailed design activity rates for all nets. This file is generated during timing simulation.
- **PWR**: This is the Power Report file which provided details of power consumption by the design

The PWR file shows the total power consumption as a split up of the baseline power (quiescent power) and the activity power.



Figure 3.8: FPGA Power Consumption Estimation using XPower

C. Development time

Development time is a significant portion of the total time to market for any application. There are number of factors which affect the development time for both hardware and software implementations. The two most important factors are a) Time to learn the tools and languages, and b) Time to implement the design using these tools and languages. There is always certain amount of “learning curve” associated with each and every tool. After the tool is learnt, the next task is to perform implementation of the design using the tools. The level of accuracy required in the design process also affects the time to complete the design. The total development time for both hardware and software implementations depends upon 3 factors:

- Number of man-hours required to perform the design
- Number of lines of code written to implement the design
- Time to learn the tools for DSP and FPGA

The time spent to implement designs in both types of implementations can be compared using these three measurements. Out of these, the time required to learn the tools is important only for first time designers. Although they do not provide an accurate measurement of the actual time spent, they can be used to have a fair comparison between both types of implementations.

Step 6: Analysis of Results

The comparison between software and hardware implementations on the basis of the three metrics described in the previous section can be illustrated with the help of the flow diagram in Figure 3.8. As execution time and power dissipation depend upon the type of DSP and FPGA chosen, it is difficult to make a generalized conclusion for all types of DSP and FPGA architectures. The flow diagram in Figure 3.9 can be used to analyze results for a particular DSP and a particular FPGA (chosen for the purpose of implementation), and not all DSPs and FPGAs in general. Time to Market is an important factor of consideration, but it cannot be used to take a decision regarding the method of implementation (hardware or software). A brief explanation of all the blocks in the flow diagram is given below:

- Compare the execution times of all software implementations and hardware implementations.
- Select the implementation with the best execution time.
- Compare the power consumption of all software implementations and hardware implementations.

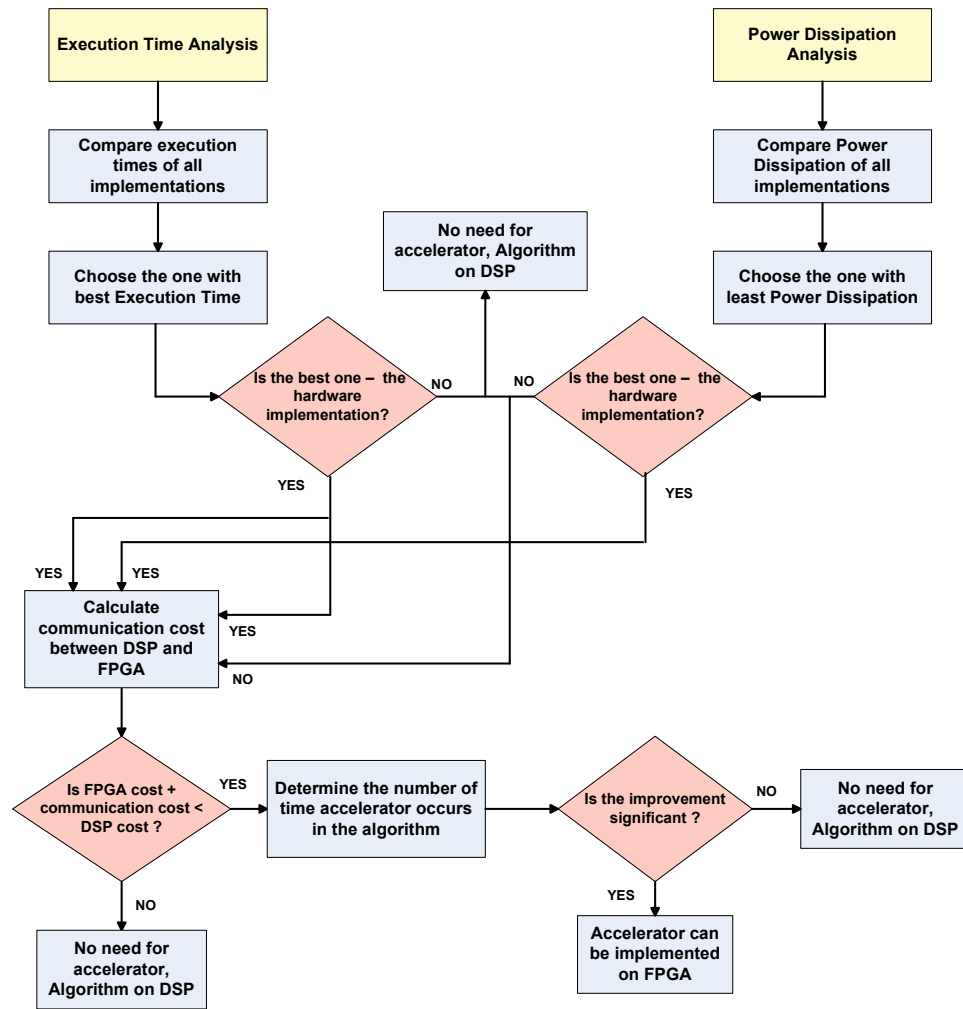


Figure 3.9: Decision Making Flow

- Select the implementation with least power consumption.
- If the software implementation of the accelerator has the best execution time and the least power consumption, there is no need for an accelerator and the entire algorithm can be implemented on DSP.
- If the hardware implementation of the accelerator has the best execution time and least power consumption, the next step is to calculate the cost of communication between FPGA and DSP. The sum of the hardware execution time and the time required for overhead communication should be less than the software

execution time.

- If the sum of the hardware execution time and the time required for overhead communication is less than the software execution time, the next step is to determine the number of times the accelerator component occurs in the algorithm.
- Observe whether the improvement obtained by implementing the accelerator in hardware is significant or not. This can be explained with the help of following example:

Hardware execution time = 20 ns

Software execution time = 400 ns

Overhead time = 300 ns

Hardware execution time + Overhead time = 320 ns

% Improvement over software implementation = $(400-320)/400 = 20 \%$

Decide whether the improvement observed is significant

- If the improvement is significant, implement the accelerator in hardware
- If the hardware execution time is less than the software execution time, but the power consumption in hardware is more than that in software, calculate the cost of overhead communication and repeat the procedure described above.

In this manner, the simulink model is converted from a Platform Independent Model (PIM) to a Platform Specific Model (PSM) (software-only or hardware-software co-design).

A comparison of Time to Market for both types of implementations can be done by comparing the number of lines of code written for each type of implementation. Although, it is not very accurate, it can be used to have an approximate estimate of the time taken for both types of implementations.

It was mentioned in the beginning of this chapter that one of the important criterion for space based applications is the size of the design. The size here means

the size of the board on which the algorithm is mapped and not the actual area (in case of hardware implementation) or the amount of code and data memory (in case of software implementation). Hence, while comparing software and hardware implementations of the accelerator components, size of the design is not chosen as the metric of comparison.

This chapter described the methodology of analyzing and implementing complex algorithms for space based applications. We begin with a Platform Independent Model of the system and at the end, on the basis of analysis of the results, the platform (s) on which the algorithm should be mapped is decided. Hence, our methodology transforms the Platform Independent Model (PIM) of the system into a Platform Specific Model (PSM).

Chapter 4

FIR Filter Case Study

In step 4.1 of the proposed methodology, which is software implementation of the accelerator component, the first step is to write/generate C code for the accelerator component. A number of Computer Aided Design (CAD) tools are currently in use during system, software and hardware design of embedded systems [43]. These tools automate the design and development process, and hence reduce the development time. One such tool is Mathwork's Real-Time Workshop, a commercial software tool which supports fast prototyping, and automatic program building from Simulink models. Although development time (also referred to as Time to Launch for space based applications) is important, it should not outweigh other more important factors like Performance and Power Dissipation. The use of Real Time Workshop increases the development speed, but at a compromise of degrading the system performance. It generates C code of a Simulink model at the click of a button, but the generated code is not as efficient as a hand written code. This chapter is dedicated to a case study which proves the above mentioned point. An FIR filter is chosen for the purpose of case study, as it is one of the basic building blocks of many signal processing and control algorithms.

We use Real Time Workshop to generate C code for an FIR filter and compare its

performance with the performance of a hand written C code. The Texas Instruments Code Composer Studio profiler is used to compare the performance of the two codes. The code generated by Real Time Workshop has some other problems like poor readability and lack of compatibility with tools from other vendors. These issues are also touched upon in this chapter. After comparing the performance of two codes, we apply code optimization techniques for improving performance. Software code optimization is part of Step 4 of the methodology described in Chapter 3. We use hand written code of the FIR filter and perform different optimizations on it. The target processor chosen in Code Composer Studio is TI TMS320C6713, a 32 bit floating point DSP. Any other DSP can also be chosen for the purpose of this case study. The basic aim is to compare the performance of tool generated and manually written codes; and show different kinds of optimization techniques that can be applied to a piece of C code.

4.1 Code Comparison

This section shows the comparison of tool generated and hand-written codes using the simple example of an FIR filter. A mathematical representation of the Direct Form of an FIR filter is given by the following equation.

$$y[n] = \sum_{k=0}^{m-1} h[k]x[n-k]$$

A sixth order direct form FIR filter is chosen for the purpose of case study. Figure 4.1 shows the structure of the FIR filter used.

A C Code for this FIR filter is written and compiled using the built in compiler provided by TI Code Composer Studio. A sequence of 40 input samples is given to the filter to examine the number of C6713 processor clock cycles consumed in the filtering operation. The C code for this simple implementation consists of 2 *for* loops. The inner loop is used to iterate the multiply-accumulate (MAC) operation for a

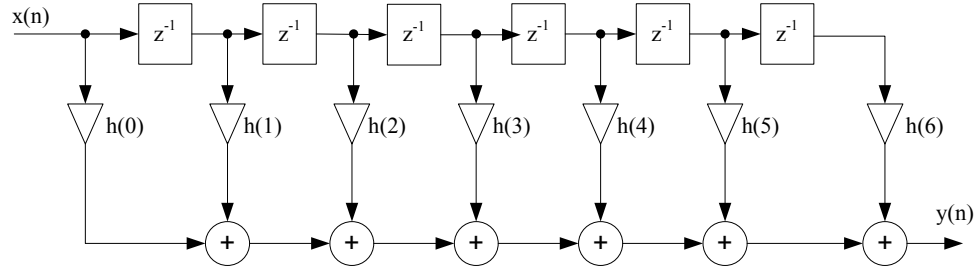


Figure 4.1: Direct Form Implementation of a 6th order FIR filter

particular output sample, and the outer loop is used to iterate for the total number of input samples which is equal to 40 in this case. All the optimization techniques described in Section 4.2.2 are applied to this basic C code. The equation written inside the inner *for* loop is as follows.

$$y[n] += x[n]h[n - m]$$

A particular output sample is calculated by passing the input sample through different delays and multiplying it with the different filter coefficients and then adding all the results. This simple implementation took 13483 processor clock cycles to process 40 input samples.

Next, we created Simulink model of this FIR filter and generated C code for it using Mathwork's Real Time Workshop. The process of generating C code from a Simulink model using RTW has been described in Chapter 2. The generated C code was imported to Code Composer Studio and compiled. The compilation of the code gave a number of errors which could not be resolved. An example of one such error is given below:

Error: Identifier mxArray not defined

Where mxArray is a data-type used by Matlab to declare arrays. The C code generated by RTW did not have any declaration of mxArray. Hence it was not

recognized by the Code Composer Studio compiler. In order to resolve the errors, a toolbox provided by MathWorks known as "Link for Code Composer" was used to generate C code and import it to Code Composer Studio. The code generated by this tool got compiled without any compilation errors. This does not mean that "Link for Code Composer" can be used to generate C code for any Simulink model. When it was used to generate C code for the Attitude Determination and Control Algorithm, it did not work well.

The C code of FIR filter generated by "Link for Code Composer" was profiled, and it was seen that it consumed about 17000 clock cycles to process 40 samples of input data which is about 1.3 times the number of clock cycles consumed by hand written code. Hence it was proved, for this example, that the C code generated using RTW and Link for Code Composer Studio was less efficient than the hand written C code. Moreover, the code generated by RTW and Link for CCS had poor readability. Figure 4.2 shows organization of both types of codes - tool generated and hand written.

The first block in Figure 4.2 shows organization of the hand written code of FIR filter and the second block shows organization of the code generated by RTW. The hand written code has just 2 functions - `main( )` and `fir_filter( )`. The tool generated code, on the other hand, has a number of function calls inside the `main( )` function. Also, all the called functions are not in the same source file; they are distributed over various files. It is very difficult to understand this type of code. Moreover, if the code does not compile, it becomes very tedious to debug the code. Table 4.1 summarizes all the facts discussed above.

This section discussed about the differences between hand written and tool generated codes. The next section deals with the different kinds of optimization techniques that can be applied to a C code to improve its performance.

<pre> void main() { // definition // declaration fir_filter() } void fir_filter() { //for loop -1 // for loop -2 } </pre>	<pre> void main() { // definition // declaration model_initialize(); model_start(); model_input(); model_terminate(); model_output(); } void model_initialize() { fir_filter_initialize(); fir_filter_input(); } void model_terminate() { fir_filter_output(); fir_filter_terminate(); } . . . </pre>
--	--

Figure 4.2: Organization of hand written and tool generated codes

4.2 Software Code Optimization

The software implementation process described by the proposed methodology (Step 4.1) mentions about optimizing the software code using different optimization techniques. The code is optimized till the point beyond which performing any optimization does not improve the performance of the code. This section discusses the different optimization techniques using the example of an FIR filter. As the tool generated code is difficult to understand, the optimization techniques are applied to the hand-written code only. At the end, we present results which show the performance improvement after performing the various optimizations.

Table 4.1: Differences between hand written and tool generated codes

<i>Hand written code</i>	<i>Tool generated code</i>
Easy to understand and debug	Difficult to understand and debug
More efficient (shown above)	Less efficient
Easy to optimize	Very difficult to optimize because of poor readability
Compatible with all compilers	May not be compatible with all compilers

4.2.1 Development flow to increase performance

Traditional development flows in the DSP industry have involved validating a C model for correctness on a host PC or Unix Workstation and then painstakingly porting the C code to hand coded DSP Assembly Language. This is both time consuming and error prone. The recommended code development flow involves utilizing the C6xxx code generation tools to aid in optimization rather than forcing you to code by hand in assembly. These advantages allow the compiler to do all the laborious work of instruction selection, parallelizing, pipelining, and register allocation. This allows focusing on getting the product to market quickly. These features simplify the maintenance of the code, as everything resides in the C framework which is simple to maintain, support and upgrade [44].

The code development flow for the TI C6xxx involves the phases described in Figure 4.3. Here the compiler is given enough information to fully maximize its

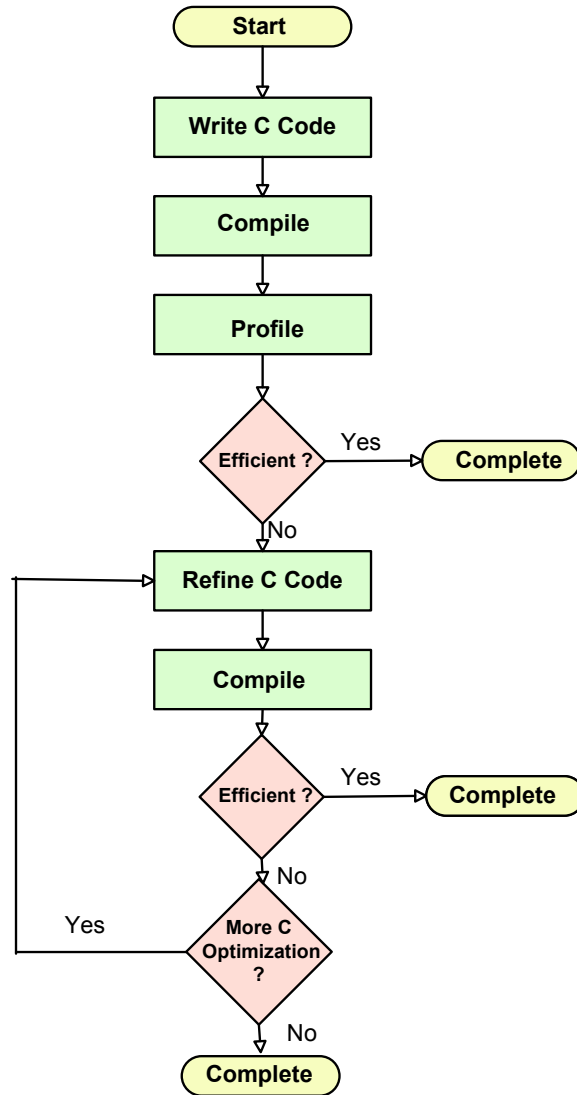


Figure 4.3: Code Development Flow for TI C6xxx

potential. The following steps summarize the flow shown in Figure 4.3.

- Write the C code, compile it, and profile it using the Code Composer Studio profiler.
- Observe the performance of the code. If the performance is matching the desired performance, stop the process.
- If the performance is not matching the desired performance, refine the code by

applying suitable optimization technique.

- Compile and Profile the refined code. Observe its performance. If it is matching the desired performance, stop the process.
- If it does not match the requirements, repeat the above steps iteratively.

For each optimization, the speed up factor will be calculated to validate its efficiency. The speedup factor can be given by the following expression:

$$Speedup = \frac{Execution_time_{old}}{Execution_time_{new}}$$

The execution time is calculated in terms of the number of processor clock cycles consumed to perform a particular operation. In the example of the FIR filter, the number of TI C6713 processor clock cycles consumed for different types of implementations is observed and then the speedup is calculated for all the types of optimizations.

4.2.2 Optimization techniques

This section describes the various optimization techniques that can be applied to a piece of C code. The hand written C code of a 6th order FIR filter is chosen as the reference standard and all the optimizations are performed on it. The number of processor clock cycles required by it to process 40 samples of input data was found to be 13483. The speedup achieved after performing each optimization will be calculated with respect to this number.

A. Register Level Optimization

The TI Code Composer studio compiler provides a number of optimization options while compiling the code. The first one of them is the Optimization -o0. This type of optimization is called Register Level Optimization. When this type of optimization option is chosen, the compiler performs control-flow-graph simplification, loop

rotation, allocates variables to registers, eliminates unused code, simplifies expressions and statements, and expands inline functions [36]. Using this optimization, an improvement is observed in the number of processor clock cycles required to process the 40 input samples by the FIR filter. The number of clock cycles now consumed is 8980. The speedup in the execution time over the previous implementation is given as

$$Speedup = 13483/8980 = 1.5$$

B. Code Clearance Optimization

The compiler optimization -o1 (code clearance optimization) is more efficient than register level optimiation. It not only performs all the optimizations performed by the register level optimization technique, but also performs local copy/constant propagation, removes unused assignments, and eliminates local common expressions [36]. It removes all the unnecessary statements from the C code. Using this optimization, an improvement is observed over the previous optimization. The number of processor clock cycles consumed for processing 40 input samples is now 7915. The speedup over the previous implementation is

$$Speedup = 13483/7915 = 1.7$$

C. Forward Store Optimization

In the C code for the FIR filter, there are 2 *for* loops and the inner *for* loop is for finding the value of a particular output sample. Initially, a global variable for the summation $y[n]$ is used. While using global variables, the compiler is obligated to perform an intervening store or load to memory during the execution of the loop. In Forward Store Optimization, [45] [37], store to a global variable $y[n]$ is done by performing the summation in a local variable *accum* and storing the final result to $y[n]$ outside the loop afterwards. This can be seen in Figure 4.4.

By doing this, the compiler understands that the local variable `accum` does not change until the next iteration of the loop, and eliminates the unnecessary loads and stores that degrade performance in the previous experiments.

<pre>for (i=0; i<4; i++) { y[n] += h[i] x[n-i]; }</pre>	$\Rightarrow$	<pre>for (i=0; i<4; i++) { accum += h[i] x[n-i]; } y[n] = accum</pre>
--	---------------	--

Figure 4.4: Code demonstrating Forward Store Optimization

The number of clock cycles to process 40 input samples after performing this optimization is 6837. So the speedup over the original optimization is given as

$$Speedup = 13483/6837 = 1.97$$

D. Loop Unrolling Optimization

The next optimization used is the compiler optimization `-o2`. It further improves performance over `-o0` and `-o1`. The optimization `-o2` performs all the `-o1` optimizations and also does software optimizations, loop optimizations and unrolling [36]. It also eliminates global common sub expressions and unused assignments, converts array references in loops to increment pointer form. Out of all these optimizations, the most efficient optimization performed by this option is Loop Unrolling.

Loop unrolling is the technique of expanding small loops so that each iteration of the loop appears in the code. This optimization increases the number of instructions available to execute in parallel. It can be used when the operations in a single iteration do not use all the resources of the C6xxx architecture. There are three ways in which loop unrolling can be performed:

1. The compiler can automatically unroll the loop
2. The compiler can be suggested to unroll the loop using the UNROLL pragma (command)
3. The C/C++ loop can be unrolled manually [44]

In the example of FIR filter, the first two techniques are adopted to perform loop unrolling to improve the execution speed. When the option (a) is used, that is, when the optimization -o2 is used, the number of processor clock cycles consumed to process 40 input samples is 5543. This shows an overall improvement over the original implementation by a factor of

$$Speedup = 13483/5543 = 2.43$$

The method (c) of performing the loop unrolling by hand is more efficient than the optimization performed by the compiler. The number of clock cycles utilized in processing 40 samples in this case is 5533. The number is close to the previous one but this proves that hand optimization is better than the compiler optimization. The loop unrolling by hand for a simple for loop which iterates 4 times is shown in Figure 4.5.

<pre>for (i=0; i<4; i++) { y[n] += h[i] x[n-i]; }</pre>		<pre>y[n] = h[1] x[n-1]; y[n] += h[2] x[n-2]; y[n] += h[3] x[n-3]; y[n] += h[4] x[n-4];</pre>
--	---	---

Figure 4.5: Code demonstrating Loop Unrolling

The third option of performing Loop unrolling is using the UNROLL pragma (a command that can be written in C code). It is written before the loop begins. If

we want to unroll the loop 4 times, then the command written above the loop is UNROLL (4). The pragma UNROLL (1) means that no loop unrolling should be performed by the compiler.

E. Combination of Forward Store and Loop Unrolling Optimizations

In this experiment, the techniques of forward store is used along with the optimization -o2 (Loop Unrolling). This further increases the execution speed, that is, decreases the number of clock cycles required for processing the 40 input samples. The number of processor clock cycles required to process the samples is now 5290. The speedup obtained in this case is

$$Speedup = 13483/5290 = 2.55$$

One more optimization is tried in this experiment. This is using the keyword register in front of the variable accum used to store the temporary value of the output sample $y[n]$. Doing this, we store the value of the variable accum in the processor register and not the external memory. But it is observed that the number of clock cycles utilized remains the same, that is, 5290. This is because that optimization is already performed by the optimization -o0.

F. Optimization using Appropriate Index Updating

In this experiment, array indexing is done in a manner that can help the compiler to translate the index arithmetic into efficient addressing. This can be shown in the fragment of code in Figure 4.6.

This change reduces the number clock cycles to half of the previous implementation. The number of clock cycles in the previous case is 5290 and now it is 2849. The speedup in this case is

$$Speedup = 13483/2849 = 4.73$$

<pre> for (i=0; i<4; i++) { accum += h[i] x[n-i]; } y[n] = accum </pre>	$\Rightarrow$	<pre> for (i=0; i<4;) { accum += h[i] x[n-i++]; } y[n++] = accum </pre>
--	---------------	--

Figure 4.6: Code demonstrating Index Updating

G. Optimization using Intrinsics

The C6000 compiler provides intrinsics, special functions that map directly to inlined processor instructions, to optimize the C/C++ code quickly. All instructions that are not easily expressed in C/C++ code are supported as intrinsics. Intrinsics are specified with a leading underscore and are accessed by calling them as we call a normal C function. The following code fragment in Figure 4.7 illustrates the use the intrinsic *_mpy*

<pre> for (i=0; i<4; i++) { accum += h[i] x[n-i]; } y[n] = accum </pre>	$\Rightarrow$	<pre> for (i=0; i<4; i++) { accum += _mpy(h[i],x[n-i]); } y[n] = accum </pre>
--	---------------	--

Figure 4.7: Code demonstrating use of Intrinsics

The use of the intrinsic *_mpy* invokes the compiler to use the MPY instruction of the C6000 processor. The MPY instruction of the C6000 processor is more efficient than a normal multiply instruction. Hence it will lead to the reduction in the number of clock cycles consumed to process the input samples. The number of processor clock cycles required to process the 40 input samples is now 1849. So there is a fall in the number of clock cycles from 13483 to 1849. The speedup achieved is

$$Speedup = 13483/1849 = 7.3$$

All the above optimization techniques were applied to the C code for the FIR filter and it is observed that optimizing the code such that it suits the processor architecture and instruction set can lead to a great improvement in the performance of the code. The following section presents all the results using tabular and graphical representation.

4.2.3 Summary

The number of TI TMS320C6713 processor clock cycles consumed to process 40 input samples and the percentage speedup achieved during the different experiments performed on the C code for FIR filter are shown in Table 4.2

Figure 4.8 shows the number of clock cycles required by each optimized code graphically. Figure 4.9 shows the percentage speedup achieved after performing each optimization graphically. It can be seen from the graphs that each optimization technique increases the execution speed of the C code. The overall speedup obtained after applying all optimizations is shown below:

$$\text{Overall Speedup} = \frac{\text{No. of clock cycles of original code}}{\text{No. of clock cycles of most optimized code}} = \frac{13483}{1849} = 7.29$$

Table 4.2: Clock cycles and Speedup for different Optimizations

<i>Optimization Technique</i>	<i>Number of Clock cycles</i>	<i>Percentage Speedup</i>
Original code	13483	BASE
Register Level Optimization	8980	34
Code Cleaning Optimization	7915	42
Forward Store Optimization	6837	50
Loop unrolling Optimization	5543	59
Manual Loop unrolling	5533	60
Loop unrolling and Forward Store	5290	61
Index updating Optimization	2849	79
Optimization using Intrinsics	1849	87

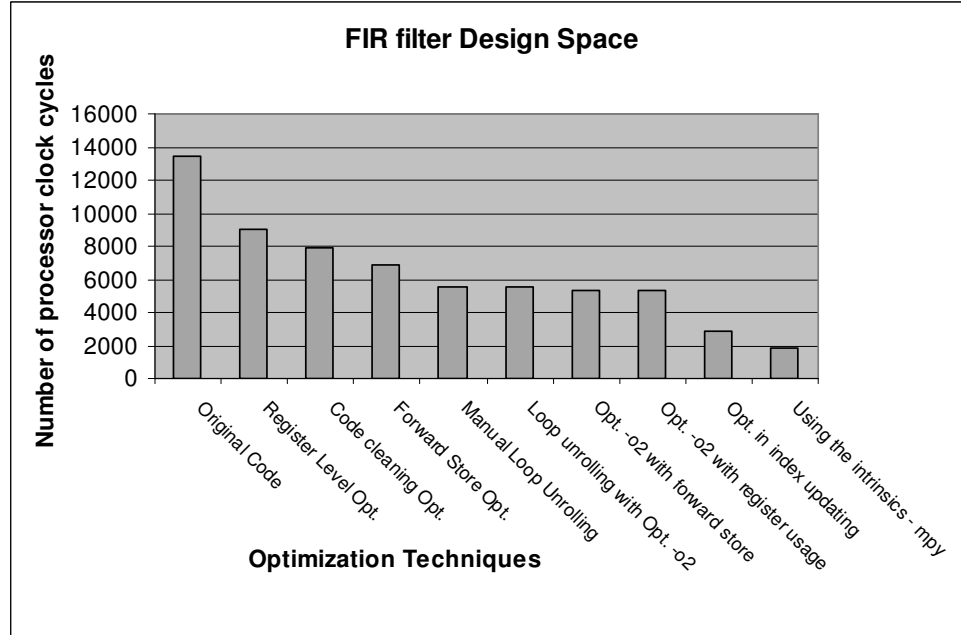


Figure 4.8: FIR filter design space

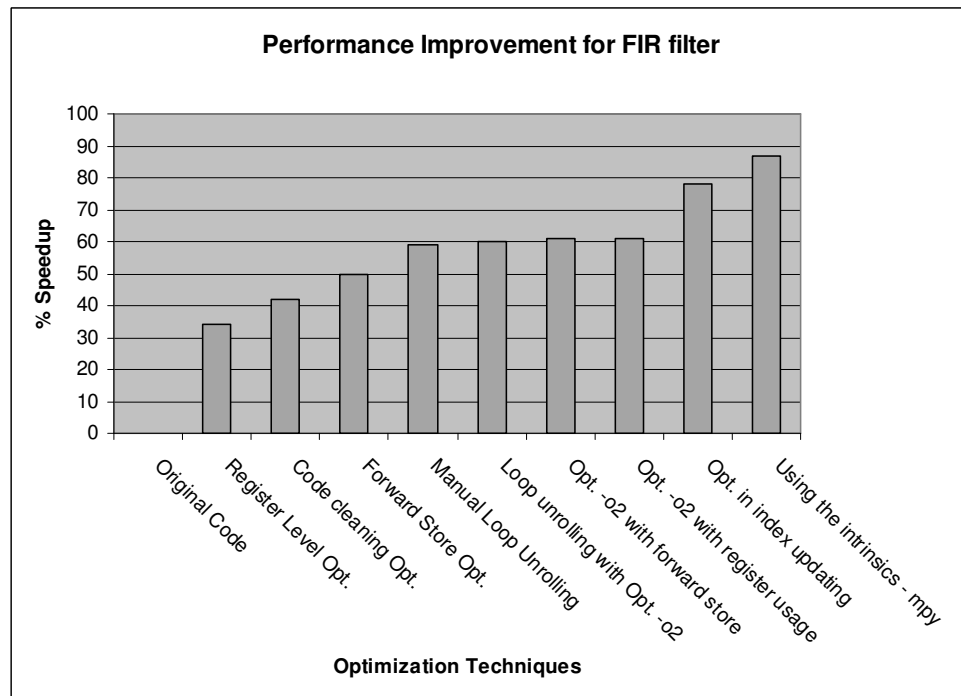


Figure 4.9: Performance Improvement for FIR filter

Chapter 5

Validation of Proposed Methodology

This chapter deals with validation of the methodology proposed in Chapter 3. This is done by analyzing the Attitude Determination and Control algorithm using the proposed methodology. The chapter is divided into three sections. The first section gives a brief overview of the ADC algorithm, the second section discusses the step by step process of analyzing the algorithm, and the final section concludes the chapter by discussing the results obtained from algorithm analysis.

5.1 Overview of ADC Algorithm [1]

The Attitude Determination and Control (ADC) algorithm has been developed as an attempt to enable small satellites to perform autonomous in space operations. This algorithm is being implemented on a controller as part of a testbed that emulates a micro-satellite with 3 Degrees of Freedom (DOF) attitude control (Roll, Pitch and Yaw) and 2 Degrees of Freedom along the horizontal plane. The testbed (which is also

known as Attitude Control System) consists of four single gimbal control momentum gyro (CMG) clusters in a pyramid configuration. The configuration can be seen in Figure 5.1. Each gimbal has a stepper motor for rotation of a DC motor with constant angular velocity of the flywheel. The testbed has two types of sensors - the rotary encoders (one on each gimbal) which give information about the angular rate of the gimbals, and the Inertial Measurement Unit (IMU), at the center of mass of the entire system, which gives information about the angular velocity of the spacecraft.

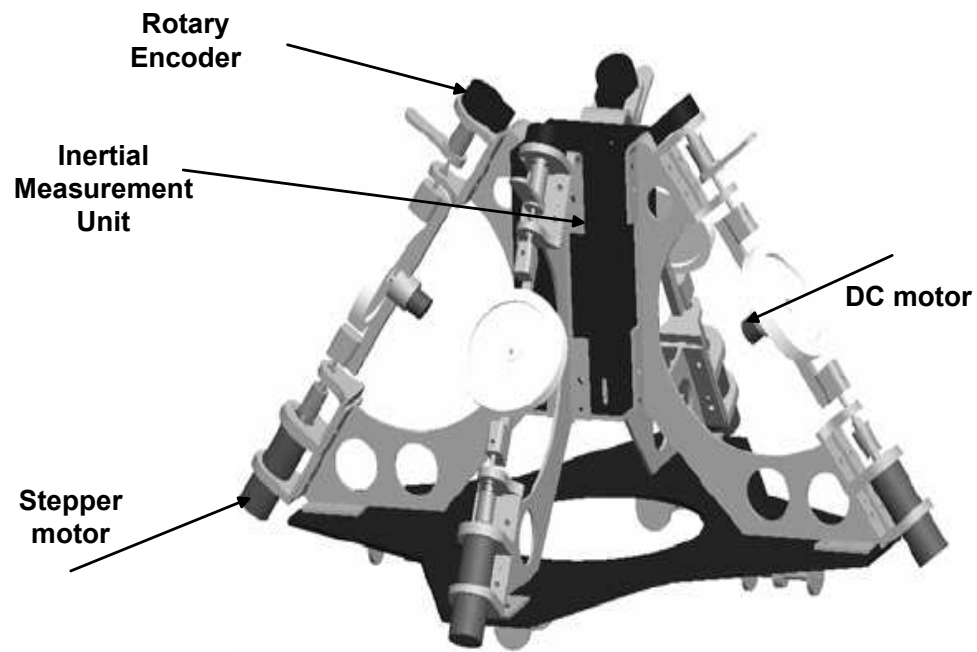


Figure 5.1: 4 SGCMG cluster in a pyramid configuration

The ADC algorithm designed by students at the AMAS laboratory will reside in a controller to be placed at the center of the system shown above. The main purpose of the algorithm is to provide inputs to the actuators (stepper motors) and get feedback from the sensors (IMU and rotary encoders). Hence the algorithm will be used for close loop control of the Attitude Control System (ACS) shown above. The high level block diagram of the algorithm is shown in Figure 5.2. The algorithm is divided into

three main parts. A brief description of each part follows the block diagram.

As a part of the development process, the algorithm has been designed and simulated by a system engineer using Simulink from Mathworks. We have been given the task of mapping the algorithm on an appropriate hardware architecture using the Simulink model provided to us. In the process of giving an overview of the model, we have mentioned certain terms used by the system level engineer. We do not go into the details of those terms, as our aim is to analyze the algorithm on the basis of the type of operations performed by it.

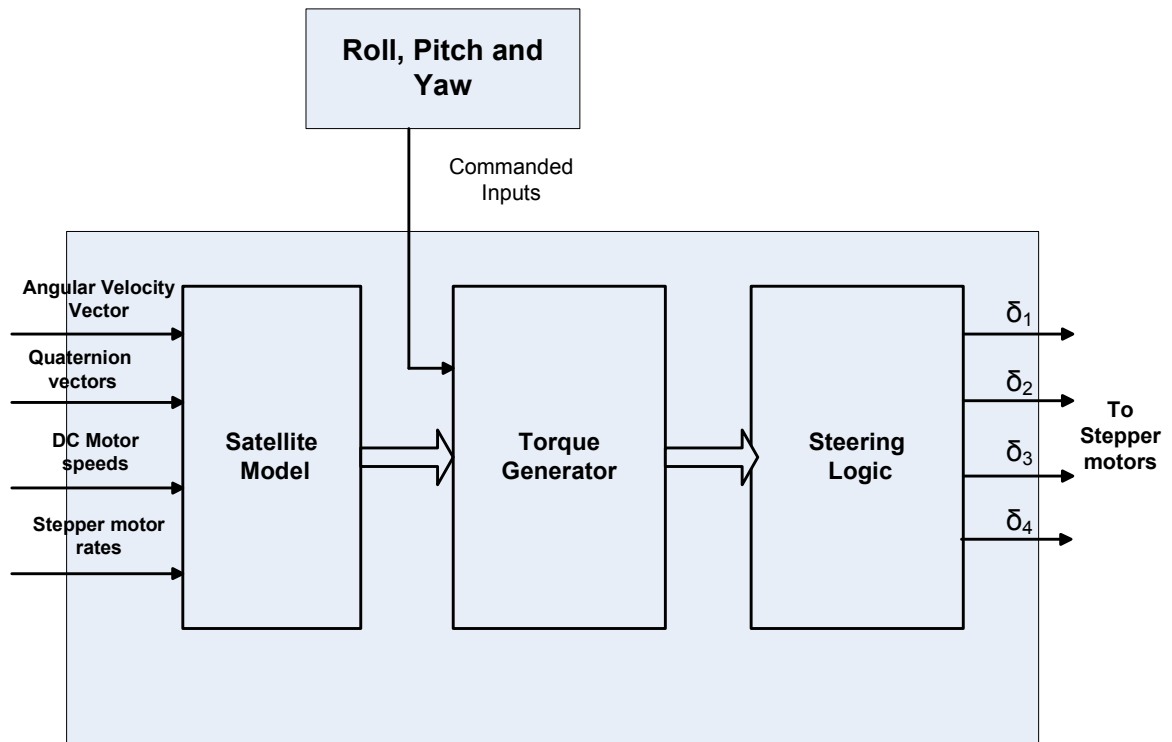


Figure 5.2: High Level Block diagram of ADC algorithm

Satellite Model

This part receives the main inputs of the algorithm. It receives the spacecraft angular velocity vector from the IMU. It also receives the stepper motor rate feedback from the 4 rotary encoders. The third input is the vector of DC motor speeds. The only function of this part is to calculate the quaternion states. It calculates these vectors by multiplying the angular velocity vector with the quaternion vector. The rest of the inputs are just passed to the next stage. So, the only significant operation that takes place in this block is of matrix multiplication.

Torque Generator

This block produces the 3 element torque vector. The torque is produced from the following equation obtained from [1]:

$$\tau = -J\{2k[sat(e + \frac{1}{T} \int e)] + c\omega\}$$

where τ is the torque, J is a 3×3 inertia matrix is fed into the system as constant, k and c are just scalar gains. T is the scalar time constant, and e is the quaternion error vector obtained by multiplying the commanded input matrix (obtained by processing the Roll, Pitch and Yaw commands) with the quaternion state matrix obtained from the previous block (Satellite Model). The integral symbol is that of numerical integration and sat is a dynamic saturation function. The output of this block is the $[3 \times 1]$ torque vector.

Steering Logic

The use of Control Momentum Gyros (CMG) in the ACS system necessitates the development of CMG steering logic, which generates the CMG gimbal rate commands for the commanded spacecraft control torques. The steering logic has to be used in

this system in order to prevent the singularity condition. The singularity condition occurs anytime the configuration of the gimbal angles affects the amount of torque that can be produced leaving the output torque to zero. The inputs to these blocks are the outputs of the *Satellite Model* and the torque vector produced by the *Torque Generator*. This block uses the Steering law described in [46] to produce the gimbal angle rates which are given to the 4 stepper motors of the Attitude Control System shown in Figure 5.1.

This is the overview of the Attitude Determination and Control algorithm. The analysis of this algorithm is carried out using the methodology proposed in Chapter 3. The following section describes the step by step process of analyzing this algorithm. Here Step X corresponds to the steps of the proposed methodology in Chapter 3.

5.2 Analysis of ADC algorithm

As mentioned above, the Attitude Determination and Control algorithm has to reside in a controller or computational processing element as shown in Figure 5.1. The analysis of the algorithm using the methodology proposed in Chapter 3 will make the task of implementing this algorithm simple. We have analyzed the algorithm in a step by step manner providing necessary details at each and every step.

5.2.1 Step 1: Choose candidate platforms

Step 1 of the methodology is to narrow down the selection of the candidate controller or processing element (PE) platforms for implementing the algorithm. Out of the four architectures - General Purpose Processor, Digital Signal Processor, Field Programmable Gate Array and Application Specific Integrated Circuit, two architectures - DSP and FPGA have been chosen as candidates for implementing the algorithm. The justification for this can be given with the help of the same example as the one illustrated in Chapter 3 where the weights given to different criteria

like Performance, Power Dissipation, Time to Market, etc, are used to determine the weighted scores for the four architectures mentioned above. We start with the assumption that the computationally intense parts of the algorithm can be implemented on an FPGA and the rest of the algorithm can be implemented on a DSP. This is because DSP has the highest total score and FPGA has the second highest total score, as seen in Figure 3.2.

5.2.2 Step 2: Algorithm analysis

Step 2 of the methodology deals with analysis of the algorithm. The algorithm has been designed using Simulink from Mathworks. We have presented a step by step process of analyzing the algorithm in Step 2 of the methodology. The following points show the application of those steps to the given Simulink model:

- RTW from Mathworks was used to generate C code of the simulink model.
- The C code was imported to TI Code Composer Studio and compiled.
- The compilation of the code gave a number of errors - one of them is "Identifier mxArray not defined".
- Attempts were made to resolve the errors, but we could not succeed. The code generated by RTW was very difficult to understand.
- Because the errors in the code could not be resolved, Path B was taken in the flow diagram shown in Figure 3.3.
- This was to divide the algorithm into smaller parts and analyze each one separately. The Simulink model is divided into 3 different modules - Satellite Model, Torque Generator and Steering Logic. We performed Time Complexity analysis on each module. The details of the complexity analysis are presented below.

In performing time complexity analysis of the each module, time complexity of each block of the module was determined. The Satellite Model has small number of

Simulink blocks, so its complexity analysis can be shown with the help of a block diagram. The other two modules, i.e., Torque Generator and Steering Logic, are very large in size, so it is not possible to show their complexity analysis with the help of figures. For these modules, we have provided tables which give information about the types of Simulink blocks they have and their time complexities. Figure 5.3 shows the Satellite Model with all the blocks replaced by their time complexities. Tables 5.1 and 5.2 show the different simulink blocks along with their time complexities for sub-models II and III respectively.

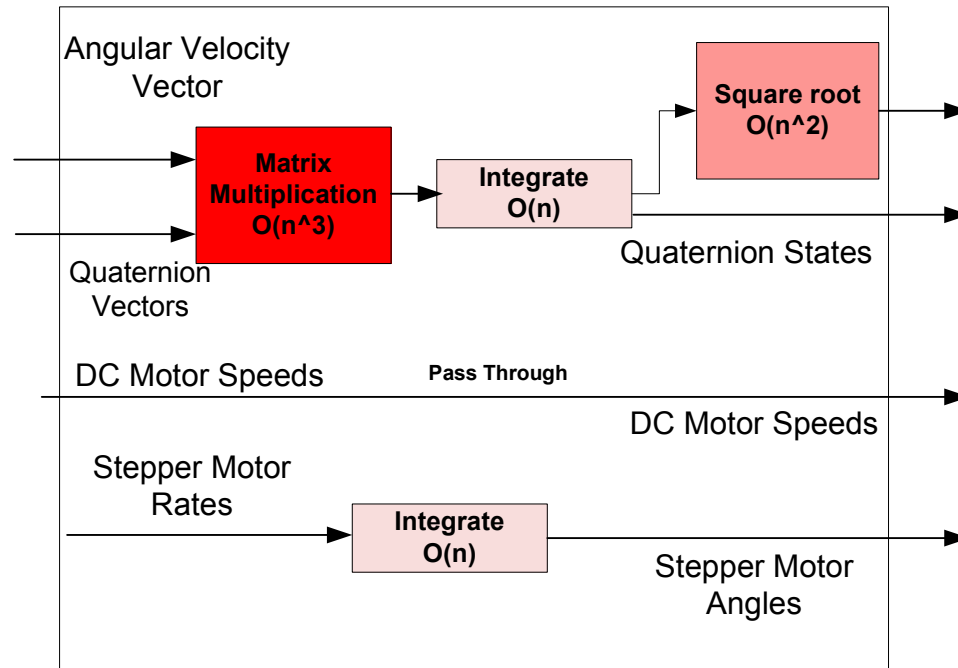


Figure 5.3: Complexity analysis of Satellite Model

The first sub-model consists of a matrix multiplication block, two integrators and a block which calculates square root of a signal. Time complexities of all these blocks have been shown in Figure 5.3. Matrix multiplication has time complexity of $O(n^3)$. This can be seen in the code in Figure 5.4. As the value of n increases, the amount of time required to multiply two matrices increases by the factor of n^3 . It can be seen

in the code that all the 3 *for* loops iterate n times.

```
for (i=0;i<n;i++)
{
    for (j=0;j<n;j++)
    {
        for (k=0;k<n;k++)
        {
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
        }
    }
}
```

Figure 5.4: Matrix multiplication code

The integrator has time complexity of $O(n)$. Integration in continuous time means accumulation in discrete time. To add n numbers, a *for* loop can be used which iterates n times. As the value of n increases, the amount of time required to add the numbers increases proportionately. There is a direct proportionality between the two. The third block in this sub-model is a square root block. The time complexity of the square root block is $O(n^2)$ using Newton's method [47]. Likewise, time complexity analysis was performed for all blocks in the other two modules. The tables show the time complexities of the various blocks used in these two sub-models. Many of the blocks occur multiple times in these two modules.

5.2.3 Step 3: Choose candidate accelerators

Step 3 of the methodology is to choose the candidate accelerator components from the results of Time Complexity analysis. This can be done in two different ways depending on the path chosen in Figure 3.3. We have chosen Path B in Figure 3.3, so Part B(time complexity analysis), described in Step 3 of the methodology, was used to find out the candidate accelerator components. The set of rules given in Step 3 were used to compare the time complexities of the various blocks. After comparing the time

Table 5.1: Time Complexity of blocks of Torque Generator

<i>Simulink Block</i>	<i>Time Complexity</i>
Matrix multiplication	$O(n^3)$
Square root	$O(n^2)$
Addition	$O(1)$
Multiplication	$O(1)$
Operation of finding minimum	$O(n)$

Table 5.2: Time Complexity of blocks of Steering Logic

<i>Simulink Block</i>	<i>Time Complexity</i>
Sine	$O(n^2)$
Cosine	$O(n^2)$
Differentiation	$O(n)$
Integration	$O(n)$
Matrix Multiplication	$O(n^3)$
LU decomposition	$O(n^3)$

complexities of all the blocks, the ones with highest time complexities were chosen as candidate accelerators. Figure 5.5 shows the selection of accelerator components. It can be seen that matrix multiplication and LU decomposition have the highest time complexities of all the blocks. Hence they were chosen as candidate for hardware accelerator components.

Satellite Model

<i>Simulink Block</i>	<i>Time Complexity</i>
Matrix multiplication	$O(n^3)$
Square root	$O(n^2)$
Integration	$O(n)$



**Matrix
multiplication**

Torque Generator

<i>Simulink Block</i>	<i>Time Complexity</i>
Matrix multiplication	$O(n^3)$
Square root	$O(n^2)$
Addition	$O(1)$
Multiplication	$O(1)$
Operation of finding minimum	$O(n)$



**Matrix
multiplication**

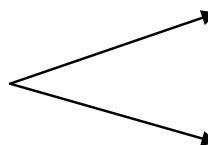
Steering Logic

<i>Simulink Block</i>	<i>Time Complexity</i>
Sine	$O(n^2)$
Cosine	$O(n^2)$
Differentiation	$O(n)$
Integration	$O(n)$
Matrix Multiplication	$O(n^3)$
LU decomposition	$O(n^3)$



**Matrix
multiplication
&
LU
decomposition**

**CANDIDATE
ACCELERATORS**



**Matrix
multiplication**

**LU
decomposition**

Figure 5.5: Selection of Candidate Accelerators

5.2.4 Step 4: HW/SW implementation of accelerators

Step 4 of the methodology is to perform software and hardware implementations of the accelerator components. We have divided this section into two parts. The first part discusses the software and hardware implementations of 3×3 Matrix Multiplication, and the second part deals with software and hardware implementations of LU decomposition of a 3×3 matrix.

I. Matrix Multiplication

A. Software Implementation of Matrix Multiplication

A C code for 3×3 matrix multiplication was written and profiled using the TI Code Composer Studio profiler. TI TMS320C6713 was the processor chosen as the target processor in Code Composer Studio. It is a 32-bit floating point processor with the VelociTi architecture. TMS320C64xx is a series of fixed point DSPs of the same family. The reasons for choosing a floating point processor instead of a fixed-point DSP are listed below:

- C/C++ does not support fixed point formats. Modeling of a bit-true fixed point system in C/C++ is difficult and slow [48].
- It is difficult to find fixed point system representation that optimally maps to system model developed in floating point [48].
- When floating point C/C++ code is targeted to fixed point DSPs, the execution speed is reduced a lot, because a floating point operation is compiled into several fixed point instructions [48].
- The bit-true fixed point C code when compiled using the standard compilers, leads to an extensive use of operator overloading, templates and dynamic memory management. This renders the fixed point implementation inefficient as compared to floating point implementations [48].

The C code was then optimized using the optimization techniques discussed in Chapter 4. The original code consisted of 3 nested *for* loops as seen in Figure 5.4. The number of processor clock cycles required for the execution of this small peice of code was **2306**. In order to improve the performance of this code, the following optimization techniques were applied to the code:

- Register Level optimization (-o0)
- Code clearance optimization (-o1)
- Forward Store optimization
- Optimization -o0 and Forward Store (FS)
- Optimization -o1 and Forward Store (FS)
- Loop Unrolling optimization (-o2)
- Manual Loop Unrolling - innermost loop
- Manual Loop Unrolling - the inner 2 loops
- Manual Loop Unrolling - all the 3 loops
- Use of intrinsic - `_mpy`

The explanation of each optimization technique has been given in Chapter 4. Some of the techniques mentioned above are compiler optimization techniques. These optimizations are performed by using different compiler build options. Some other optimizations like Forward Store and Loop Unrolling were performed manually on the code. Figure 5.6 shows the different types of optimizations performed on the basic C code of matrix multiplication. Performing these optimizations on the C code improves the performance of the code. Table 5.3 shows the number of processor clock cycles required to execute each optimized code along with the percentage speedup for each optimized code. If N_o is the number of clock cycles consumed by the original

code and N_n is the number of clock cycles consumed by the optimized code, then the speedup for each optimization is calculated using the formula shown below:

$$Speedup = \frac{N_o - N_n}{N_o}$$

Table 5.3: Clock cycles and % Speedup for different optimizations for Matrix multiplication

<i>Optimization Technique</i>	<i>Number of Clock Cycles</i>	<i>% Speedup</i>
Original code	2306	BASE
Forward Store Optimization	2038	11.6
Optimization <code>-o0</code> and F.S.	1482	35.7
Optimization <code>-o1</code> and F.S.	1176	49
Optimization <code>-o2</code> and F.S.	457	80.18
Unrolling innermost loop	662	71.29
Unrolling 2 inner loops	324	85.9
Unrolling all the loops	208	90.98
Use of <code>_mpy</code>	102	95.57
Use of <code>reg</code> keyword	94	96

It can be seen from Table 5.3 that the least number of clock cycles required to execute the C code of 3×3 matrix multiplication is **94** with a speedup of **96 %** over the performance of original code. This is the best performance that we could achieve in software implementation of matrix multiplication. We now describe the hardware implementation of matrix multiplication on Xilinx Virtex-4 FPGA.

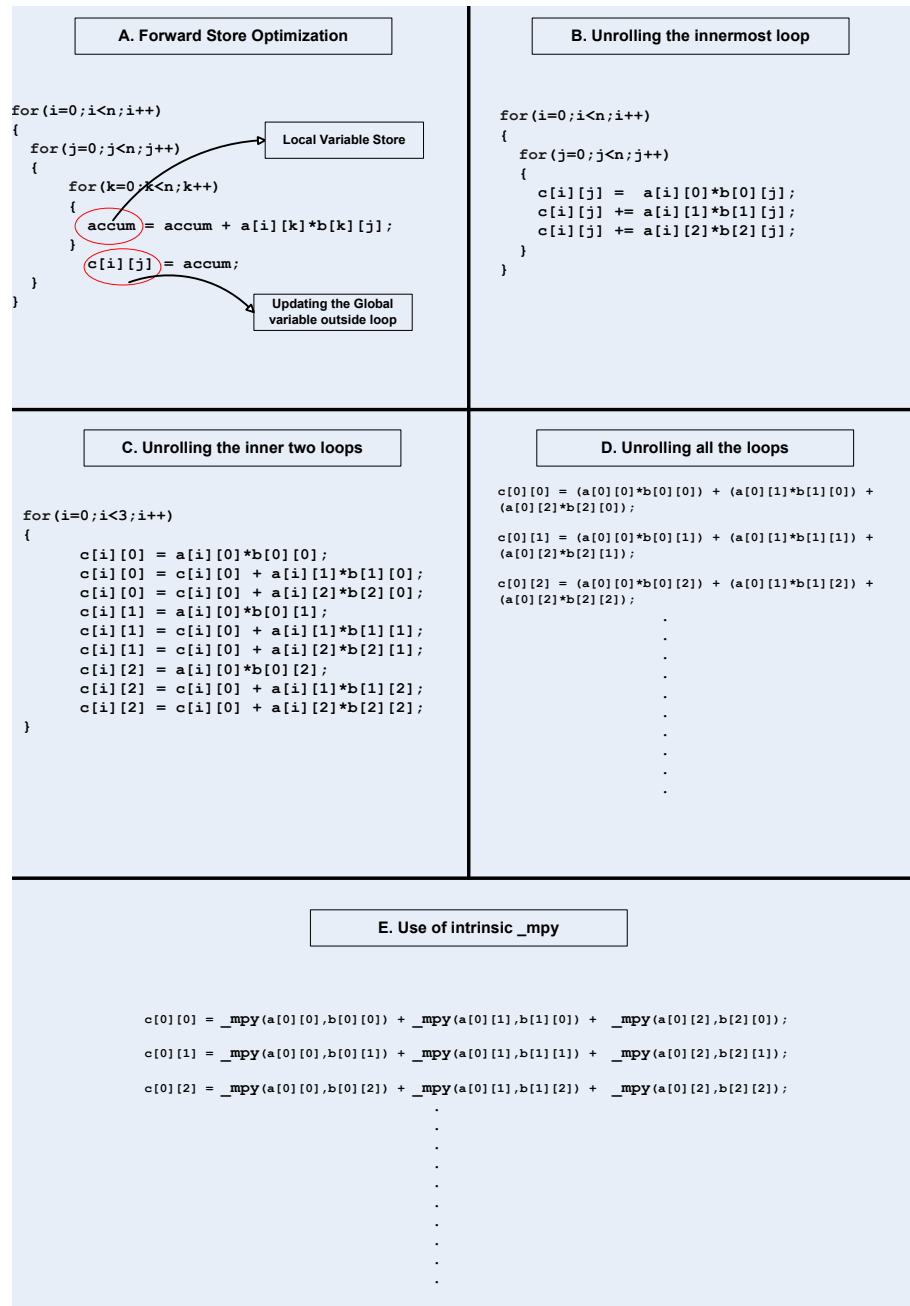


Figure 5.6: Optimizations performed on C matrix multiplication code

B. Hardware Implementation of Matrix Multiplication

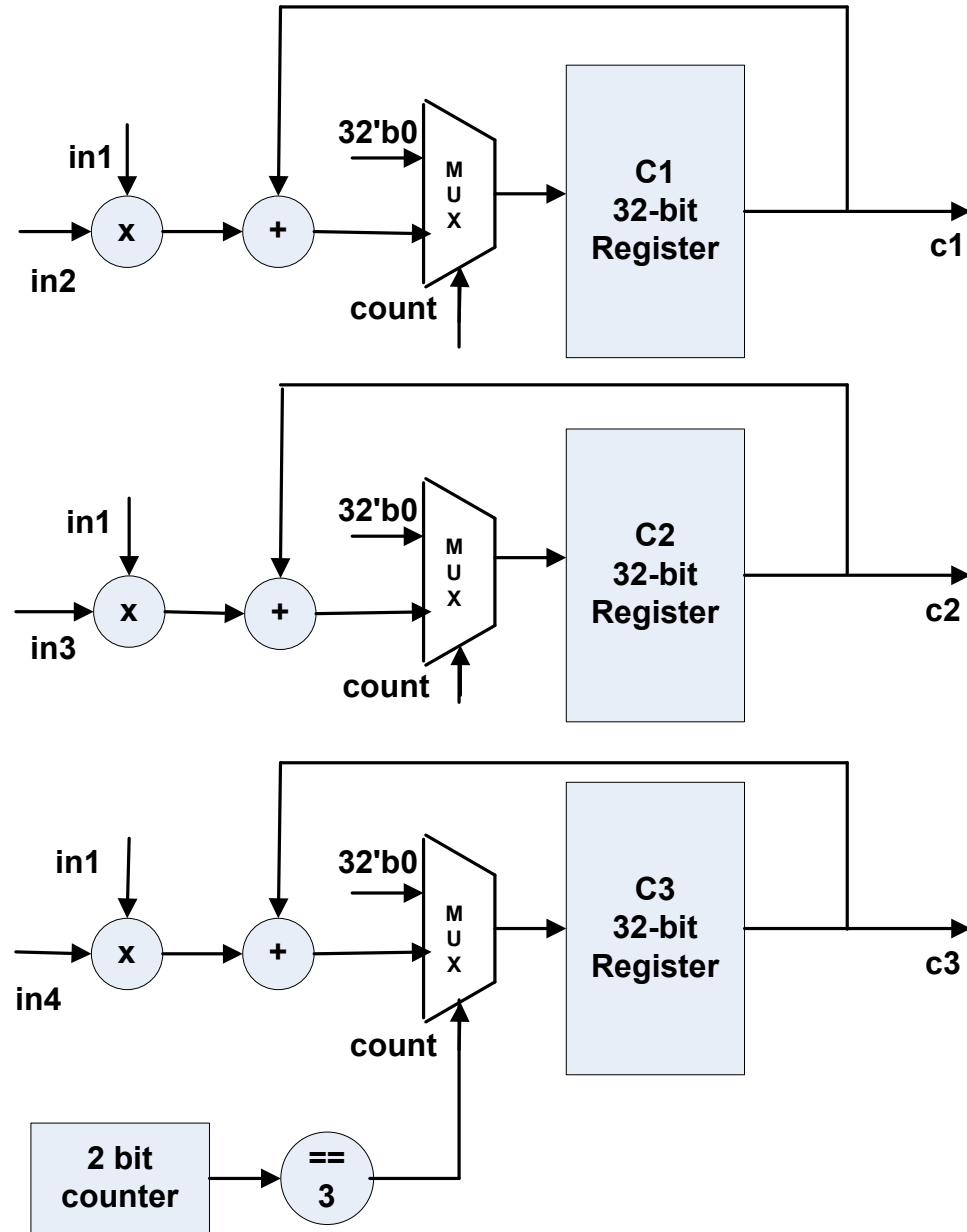
Figure 5.7 shows the logic diagram of hardware implementation of 3×3 matrix multiplication. There are 4 32-bit inputs and 3 32-bit outputs. Let us say that the matrices to be multiplied are denoted by A and B. The elements of matrix A are sent through the input *in1*, and the elements of matrix B are sent through inputs *in2*, *in3* and *in4*. The elements of the resultant matrix come out of the outputs *c1*, *c2* and *c3*. The *count* signal which goes high every time the output of the 2-bit counter equals 3. When count is equal to 1, the registers *c1*, *c2* and *c3* are cleared. The sequence in which inputs are given to the system can be shown below:

```

in1 - > a11; in2 - > b11; in3 - > b12; in4 - > b13
in1 - > a12; in2 - > b21; in3 - > b22; in4 - > b23
in1 - > a13; in2 - > b31; in3 - > b32; in4 - > b33
in1 - > a21; in2 - > b11; in3 - > b12; in4 - > b13
in1 - > a22; in2 - > b21; in3 - > b22; in4 - > b23
in1 - > a23; in2 - > b31; in3 - > b32; in4 - > b33
in1 - > a31; in2 - > b11; in3 - > b12; in4 - > b13
in1 - > a32; in2 - > b21; in3 - > b22; in4 - > b23
in1 - > a33; in2 - > b31; in3 - > b32; in4 - > b33

```

A verilog code for the design shown in Figure 5.7 was written and imported to Xilinx ISE. The design was simulated, synthesized and implemented on Virtex-4 FPGA according to the procedure described in Step 4.2 of the proposed methodology. The results obtained after performing the hardware implementation are shown in Table 5.4. Here *Slice Utilization* refers to the percentage of logic slices utilized by the design on the target FPGA. The inputs and outputs of the design were chosen to be of 32-bit width because the performance of this design has to be compared with the performance of software implementation of the design. The C code for matrix multiplication was targeted for a 32-bit DSP and so the inputs and outputs of the

Figure 5.7: Hardware Implementation of 3×3 matrix multiplication

hardware design were chosen to be of 32 bits in order to have a fair comparison between the two implementations.

Table 5.4: Hardware Implementation results for matrix multiplication

Number of simulation clock cycles	9
Max. Design Frequency after synthesis	348.482 MHz
Max. Design Frequency after Place and Route	389.406 MHz
Slice Utilization	1% (48 out of 6144)

II. LU decomposition

LU decomposition is used in solving a system of linear equations. Consider the system of linear equations shown below:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \dots + a_{1N}x_N = b_1$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \dots + a_{2N}x_N = b_2$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \dots + a_{1N}x_N = b_3$$

This can be written in matrix form as

$$Ax = b$$

where $\mathbf{A}$ is the matrix of coefficients and $\mathbf{b}$ is the right hand side written as column vector. The process of decomposing the matrix $\mathbf{A}$ into two matrices $\mathbf{L}$ (lower triangular matrix) and $\mathbf{U}$ (upper triangular matrix) is known as LU decomposition of a matrix. This decomposition can be used to solve the linear set

$$Ax = (LU)x = L(Ux) = b$$

by first solving for the vector $\mathbf{y}$ such that

$$Ly = b$$

and then solving

$$Ux = y$$

This is advantageous because solution of triangular set of equations is easier than solution of rectangular set of equations [49]. In the ADC algorithm, LU decomposition is used in the Steering Logic block to calculate inverse of a 3×3 matrix. Apart from LU decomposition, there are several other techniques of matrix decomposition, e.g. QR decomposition, Cholesky decomposition and GaussJordan decomposition. Some of these decomposition techniques are more efficient than LU decomposition. In such cases, the design engineer should give feedback to the system engineer to modify the algorithm by replacing the existing blocks with more efficient blocks.

The software and hardware implementations of LU decomposition of a 3×3 matrix are described in the ensuing lines.

A. Software Implementation of LU decomposition

The software implementation of LU decomposition of a 3×3 matrix was performed such that it replaced the original matrix by its LU decomposition. The newly formed matrix was the combination of *upper triangular* and *lower triangular* matrices. The algorithm used to perform both software and hardware implementations of LU decomposition has been obtained from [50]. The algorithm used to perform the software implementation is illustrated in Figure 5.8. It can be explained with the help of an example. Let us consider a 3×3 matrix A shown below:

$$\begin{bmatrix} 0.2 & 0.4 & 0.1 \\ 0.1 & 0.1 & 0.4 \\ 0.8 & 0.5 & 0.3 \end{bmatrix}$$

It is a 3×3 matrix, so $N=3$. The following steps show the decomposition process.

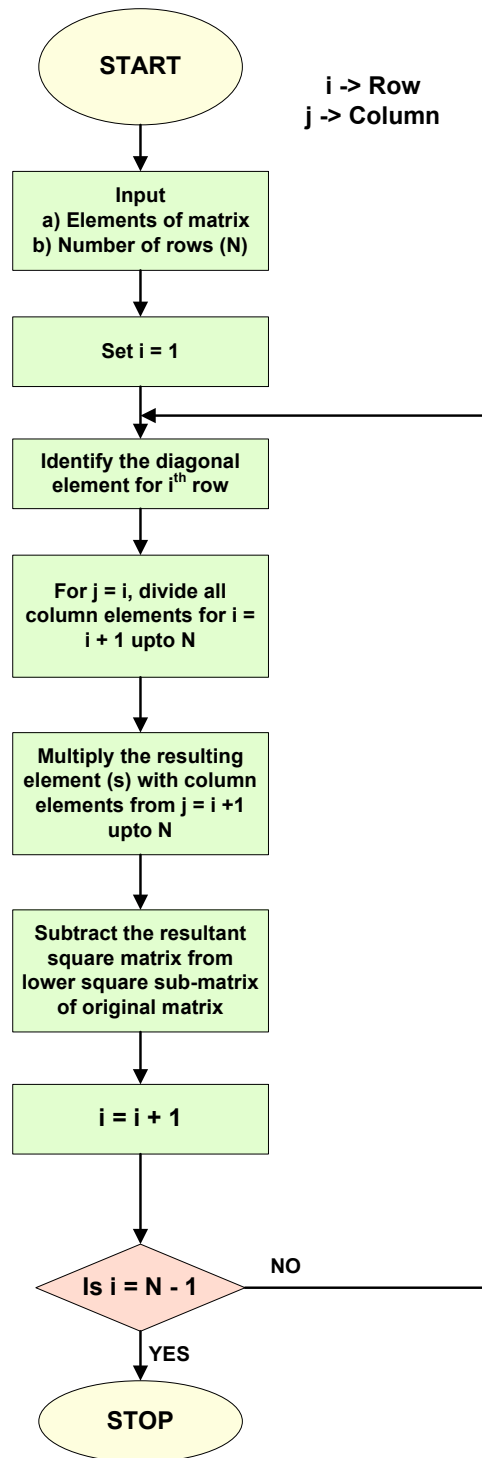


Figure 5.8: Software implementation of LU decomposition

1. $i=1$; the diagonal element of the matrix for the first row is $a_{11} = 0.2$
2. Divide a_{21} and a_{31} by a_{11} . The resulting matrix is

$$\begin{bmatrix} 0.2 & 0.4 & 0.1 \\ 0.5 & 0.1 & 0.4 \\ 4.0 & 0.5 & 0.3 \end{bmatrix}$$

3. Multiply a_{21} and a_{31} with a_{12} and a_{13} . This forms a 2×2 matrix. This is shown below

$$\begin{bmatrix} 0.5 \\ 4.0 \end{bmatrix} * \begin{bmatrix} 0.4 & 0.1 \end{bmatrix} = \begin{bmatrix} 0.2 & 0.05 \\ 1.6 & 0.4 \end{bmatrix} \quad (5.1)$$

4. Subtract this matrix from the lower sub-matrix of the original matrix, which is

$$\begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 0.1 & 0.4 \\ 0.5 & 0.3 \end{bmatrix} \quad (5.2)$$

5. The resultant matrix A is given as

$$\begin{bmatrix} 0.2 & 0.4 & 0.1 \\ 0.5 & -0.1 & 0.35 \\ 4.0 & -0.8 & -0.1 \end{bmatrix}$$

6. Now $i=i+1 = 2$; the diagonal element for this row is $a_{22} = -0.1$.
7. Divide a_{32} by a_{22} . The resulting matrix is

$$\begin{bmatrix} 0.2 & 0.4 & 0.1 \\ 0.5 & -0.1 & 0.35 \\ 4.0 & 8.0 & -0.1 \end{bmatrix}$$

8. Multiply a_{32} with a_{23} and subtract the result from a_{33} . The resulting matrix is shown below:

$$\begin{bmatrix} 0.2 & 0.4 & 0.1 \\ 0.5 & -0.1 & 0.35 \\ 4.0 & 8.0 & -2.9 \end{bmatrix}$$

9. $i=i+1 = 3$; but $i > N-1$, so we stop the process. The matrix shown above is decomposition "in place" of the original matrix A.

A C code for the algorithm shown in Figure 5.8 was written and compiled for TIC6713 processor. The functionality of the code was verified by providing 10 different 3×3 matrices and matching the outputs with the expected outputs. In order to profile the C code, a 3×3 matrix was given as input and the number of processor clock cycles required to execute the code was observed. The code took **3755** processor clock cycles to perform the decomposition process. A number of optimization techniques were applied to the code in order to improve its performance. The techniques used are shown below:

- Register Level Optimization (`-o0`)
- Code Clearance Optimization (`-o1`)
- Loop Unrolling Optimization (`-o2`)
- Optimization using Register storage
- Optimization using intrinsic `_mpy`
- Optimization using intrinsic `_fabsf`
- Optimization using intrinsic `_rcpsp`

An explanation of some of the optimizations mentioned above has been given Chapter 4. The only new optimizations used here are optimizations using intrinsics `_fabsf` and `_rcpsp`. An overview of these optimizations is given below:

- *Use of `_fabsf`*: `_fabsf` invokes the processor instruction **ABSSP** which calculates the absolute value of a number in one clock cycle. It reduces the overhead of calling and executing the function `abs()`. Hence, the use of `_fabsf` reduces the overall execution time.
- *Use of `_rcpsp`*: Since there is no hardware support for division operation on the C6713 CPU, the compiler includes a function for performing the division operation. But this function call has overhead associated with it. However the division operation can be equally performed by replacing it with a "reciprocal" compiler intrinsic (`_rcpsp`) followed by multiplication operation. The optimized code directly maps into assembly instructions and thereby avoids the function call which leads to reduction in the overall execution time.

The number of processor cycles consumed by each optimized code along with the % speedup obtained for each optimization over the original implementation are listed in Table 5.5.

Table 5.5: Clock cycles and % Speedup for different optimizations for LU decomposition

<i>Optimization Technique</i>	<i>Number of Clock Cycles</i>	<i>% Speedup</i>
Original code	3755	BASE
Optimization <code>-o0</code>	2296	38.85
Optimization <code>-o1</code>	1872	50.14
Optimization <code>-o2</code>	1346	64.15
Use of <code>reg</code>	1320	64.84
Use of <code>_mpy</code>	815	78.29
Use of <code>_fabsf</code>	751	80
Use of <code>_rcpsp</code>	717	80.9

A. Hardware Implementation of LU decomposition

The algorithm illustrated in Figure 5.8 was used to build the hardware for LU decomposition of 3×3 matrix. The high level block diagram of the architecture is shown in Figure 5.9. This architecture represents the brute-force implementation of LU decomposition of 3×3 matrix. The original matrix is stored in a register bank which acts as memory. A finite state machine is used to access the elements of the matrix stored in the register bank and pass them to the appropriate modules. The elements of the decomposed matrix are stored in another register bank as and when they are generated by the design. The fixed point format of data chosen for the purpose of implementation is Q16.16 (16 bits of integer and 16 bits of fraction). Other data formats like Q4.28, Q5.27, etc can also be selected for the purpose of implementation. The selection of data format should be such that the resultant numbers should not exceed the dynamic range of the chosen data format. The explanation of each module of the architecture is given below.

Finite State Machine I

The first Finite State Machine is used to get the 32-bit elements of the matrix to be decomposed and pass them to appropriate modules. It generates a number of control signals which are used to control the sequence of operations occurring in various modules. The operation of this FSM can be understood by looking at the state diagram shown in Figure 5.10. The inputs and outputs of this module are also shown in the Figure.

As seen in Figure 5.10, there are five states (S0 - S4) in the state diagram of FSM-I. The sequence of operations performed by it is shown below:

- In the reset condition, the FSM remains in state S0. When the input (*main_start*) is given to the FSM, it goes to state S1 and the process of decomposition of the matrix starts.
- In state S1, the FSM accesses elements a_{11} , a_{21} and a_{31} from the register bank

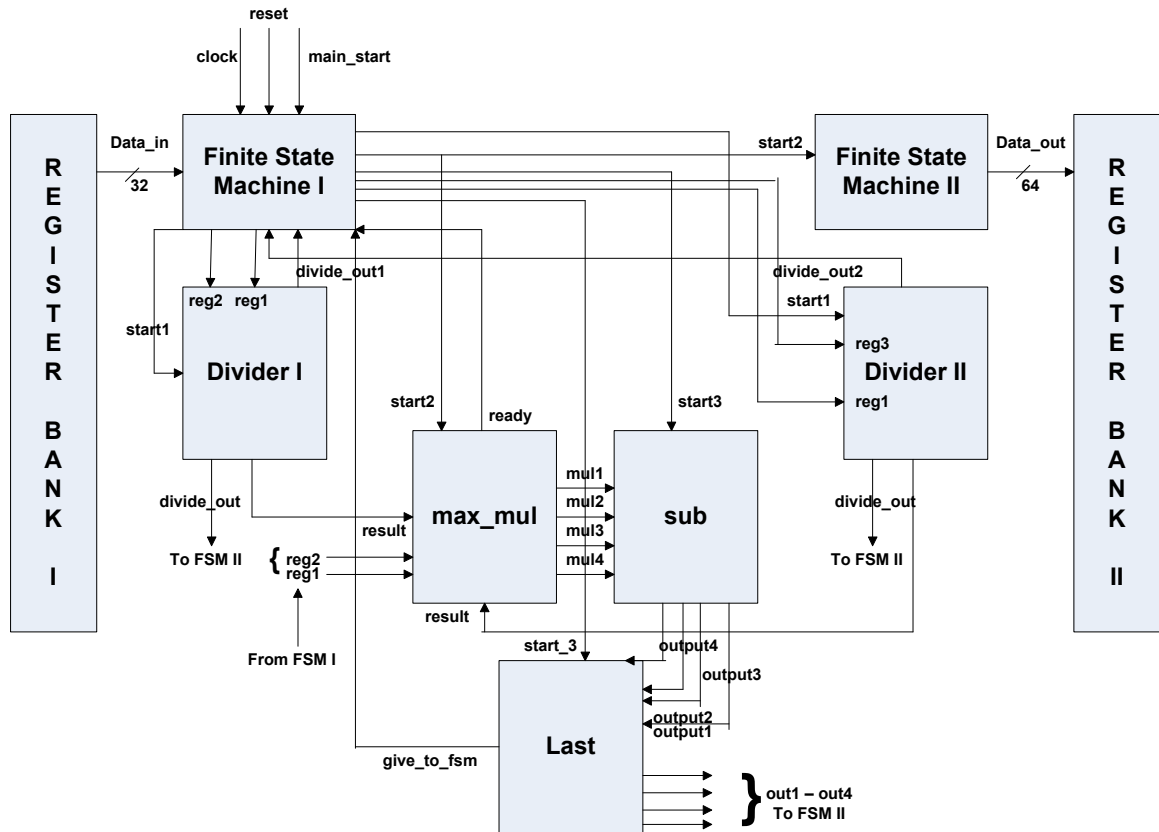


Figure 5.9: Hardware implementation of LU decomposition

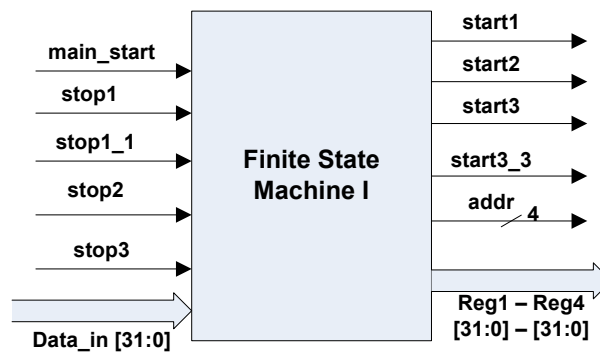
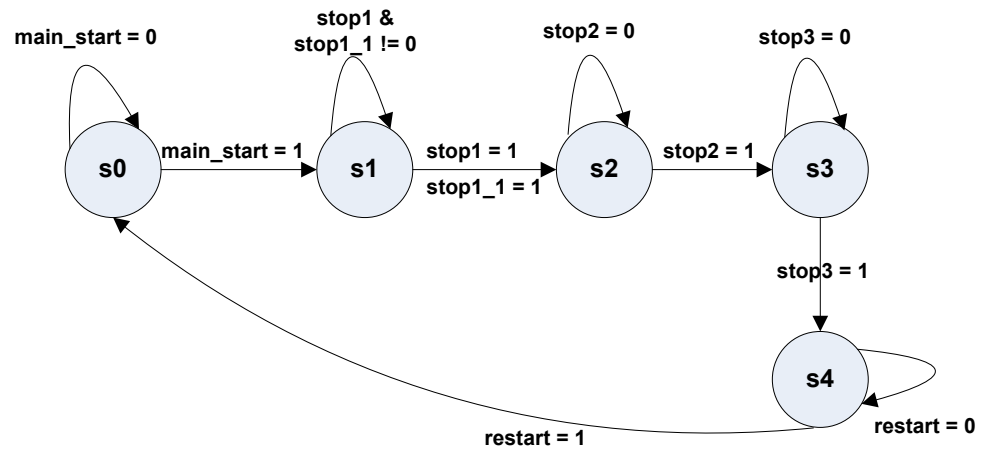


Figure 5.10: State Diagram of FSM I

and gives them to the two *divide* modules. It also gives the signals *start1* and *start1_1* to the divide modules which enable them to start the division operation. It remains in this state till it gets the signals *stop1* and *stop1_1* from the divide modules.

- When both *start1* and *start1_1* are high, the FSM goes to state S2. In this state, it accesses a_{12} and a_{13} and generates the signal *start2*. This when given to the *max_mul* module, enables the module to perform matrix multiplication.
- After matrix multiplication is over, the FSM gets a signal *stop2* and it goes to state S3. In this state, it accesses elements a_{22} , a_{23} , a_{32} and a_{33} from the register bank and passes them to the *sub* module along with the *start3*. It remains in this state till the process of subtraction is not over.
- Once the FSM receives *stop3* signal, it goes to state S4 and remains in that state. After the entire process of decomposition is over, it receives a signal *restart* which enables it to go to state S0. Upon receipt of *main_start* signal, the entire process starts again.

Divider

The division operation is the one which consumes maximum number of clock cycles in the process of LU decomposition. Although all the elements of the matrix to be decomposed are normalized (between -1 and +1), the result of division can exceed this range. For example, if the dividend is 0.8 and the divisor is 0.2, the result of division is 4.0 (which exceeds the range [-1,1]). In order to represent this result, the Q16.16 format has been selected for the purpose of implementation (as stated earlier). The hardware architecture of the division module is shown in Figure 5.11.

The inputs to this module are a) *start* b) *input_a*, and c) *input_b*. Here *input_a* is the dividend and *input_b* is the divisor. 2's complement of the inputs is taken if they are negative. In every clock cycle, the divisor is subtracted from the dividend.

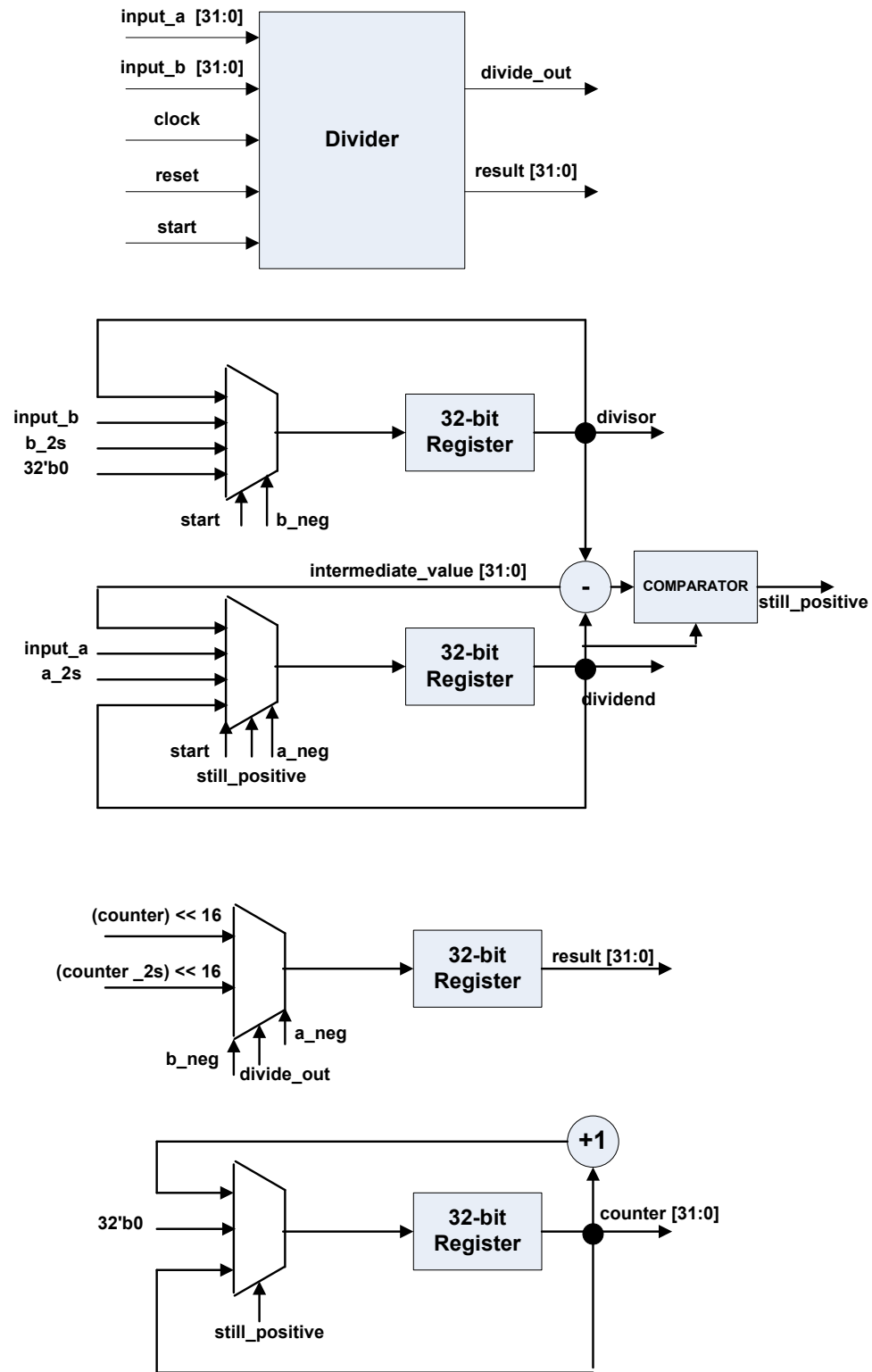


Figure 5.11: Hardware architecture of division module

If the result (*intermediate_value*) is positive, a 32-bit counter is incremented. This is repeated till the *intermediate_value* becomes negative. The value of *counter* when *intermediate_value* becomes negative is the actual result of the division operation. The value of counter is shifted to the left by 16 bits to get the appropriate result. *a_neg* and *b_neg* are signals which indicate the signs of the dividend and the divisor respectively. 2's complement of the final result is taken if either of these signals is a '1'. When the result is obtained, a signal called *divide_out* is sent out to indicate that the operation is over.

Max_mul

This module performs multiplication of 2×1 and 1×2 matrices. The inputs to this module are a_{21} , a_{31} , a_{12} and a_{13} . The output of this module is a 2×2 matrix which is sent to the module *sub* to perform the subtraction operation. The module starts its operation upon receipt of the signal *start2* from FSM-I. 2's complement of the inputs is taken if they are negative. The sign of the product is decided on the basis of the sign of the two multiplicands. The block diagram in Figure 5.12 shows the inputs and outputs of this module.

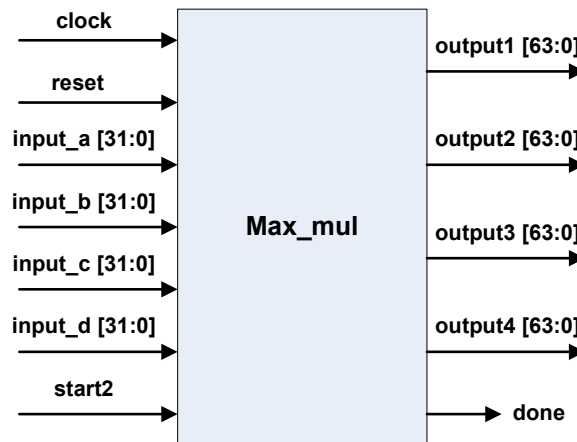


Figure 5.12: Max_mul module

Sub

This module performs the subtraction operation. It receives inputs a_{22} , a_{23} , a_{32} and a_{33} from the register bank through FSM-I. It also receives the 2×2 matrix from the module *Max_mul*. When the 2×2 matrix in module *Max_mul* is formed, it sends a signal called *done* to this module. As soon as this module receives the *done* signal, it subtracts the matrix obtained from the 2×2 matrix formed by the elements a_{22} , a_{23} , a_{32} and a_{33} . The elements of the resultant matrix are sent to the *last* module where the second level decomposition takes place. The inputs and outputs of this module can be seen in Figure 5.13.

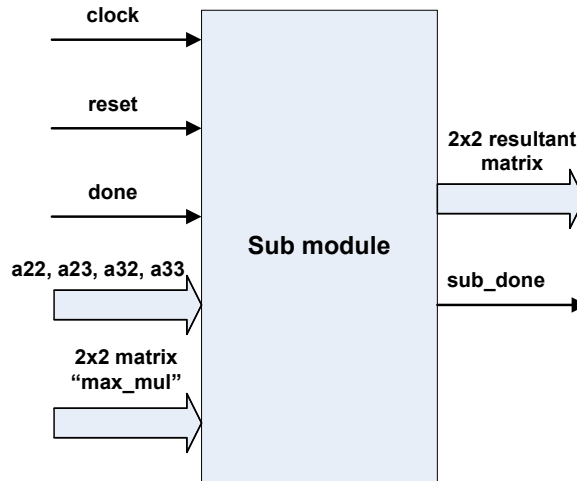


Figure 5.13: *Sub* module

The next module of the design is the one which performs second level decomposition. In the algorithm explained in Figure 5.8, it performs the operations from $i = 2$, *i.e.*, the second row. It first performs the division operation, then the multiplication operation, and finally the subtraction operation. The subtraction operation forms the last element of the decomposed matrix. The second Finite State Machine (FSM-II) receives all the elements of the decomposed matrix and sends them to the second register bank. This completes the description of the overall architecture of LU

decomposition of 3×3 matrix.

A verilog code for the design shown in Figure 5.9 was written and imported to Xilinx ISE. The functionality of the code was verified using the same inputs as used in the case of software implementation. The design was simulated, synthesized and implemented on Virtex-4 FPGA according to the procedure described in Step 4.2 of the proposed methodology. The results obtained after performing the hardware implementation are shown in Table 5.6.

Table 5.6: Hardware Implementation results for LU decomposition

Number of simulation clock cycles	33
Max. Design Frequency after synthesis	55.38 MHz
Max. Design Frequency after Place and Route	50.078 MHz
Slice Utilization	9% (584 out of 6144)

5.2.5 Step 5: Comparative Metrics measurement

This section deals with measurement of Execution Time, Power Dissipation and Development time for both software and hardware implementations and have a comparison between the two implementations on the basis of these metrics. The metrics are measured using the procedure illustrated in Step 5 of the proposed methodology. We have divided this section into three parts. The first part deals with measurement of Execution Time for software and hardware implementations for both matrix multiplication and LU decomposition. The second section discusses measurement of Power Dissipation for software and hardware implementations for both the accelerators. In the next section, we analyze the results and take a decision regarding the implementation of the algorithm.

Execution Time

Matrix Multiplication

The software implementation of matrix multiplication has been done with TI TMS320C6713 as the target processor. This processor operates at a maximum frequency of 225 MHz. The execution time for a particular implementation can be obtained by dividing the number of processor clock cycles required to execute the code by the maximum clock frequency at which the processor operates. Table 5.7 shows the execution time for each software implementation of a 3×3 matrix multiplication.

Table 5.7: Execution Times for Software Implementations of Matrix Multiplication

<i>Implementation Type</i>	<i>No. of clock cycles</i>	<i>Execution Time (ns)</i>
Original code	2306	10248.9
Forward Store Optimization	2038	9057.78
Optimization <code>-o0</code> and F.S.	1482	6586.67
Optimization <code>-o1</code> and F.S.	1176	5226.67
Optimization <code>-o2</code> and F.S.	457	2031.11
Unrolling innermost loop	662	2942.22
Unrolling 2 inner loops	324	2140
Unrolling all the loops	208	924.44
Use of <code>_mpy</code>	102	453.33
Use of <code>reg</code> keyword	94	417.77

The calculation of execution time for hardware implementation of Matrix Multiplication is shown below:

$$\text{Execution Time} = \frac{\text{No.of simulation clock cycles}}{\text{Max.design frequency after PAR}} = \frac{9}{389.406\text{MHz}} = 23.11\text{ns}$$

LU decomposition

The software implementation of matrix multiplication has been done with TI TMS320C6713 as the target processor. Table 5.8 shows the execution time for each software implementation of LU decomposition of a 3×3 matrix.

Table 5.8: Exection Times for Software Implemetations of LU decomposition

<i>Implementation Type</i>	<i>No. of clock cycles</i>	<i>Execution Time (ns)</i>
Original code	3755	16688.88
Optimization -o0	2296	10204.44
Optimization -o1	1872	8320
Optimization -o2	1346	5982.22
Use of <i>reg</i>	1320	5866.67
Use of <code>_mpy</code>	815	3622.22
Use of <code>_fabsf</code>	751	3337.77
Use of <code>_rcpsp</code>	717	3186.67

The calculation of execution time for hardware implementation of LU decomposition is shown below:

$$\text{Execution Time} = \frac{\text{No.of simulation clock cycles}}{\text{Max.design frequency after PAR}} = \frac{33}{50.078\text{MHz}} = 658.97\mu s$$

Power Dissipation

For software implementation, power dissipation measurement has been done using the Power Estimation Spreadsheet for TMS320C6713 provided by Texas Instruments. In order to have a fair comparison of power dissipation between DSP (software) and FPGA (hardware) designs, the measurement of power dissipation for DSP has been done taking into account that the power is getting dissipated due to activity level changes only in the CPU core of the DSP and not any other module. The concept is illustrated in Figure 5.14.

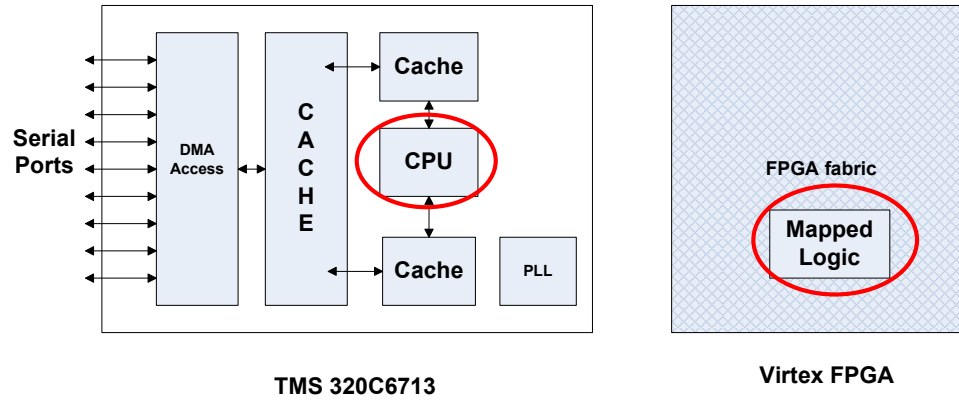


Figure 5.14: Diagram to illustrate concept of power dissipation calculation

The DSP has a number of peripheral components surrounding the CPU core. All these components operate at fixed frequencies. A certain amount of power gets dissipated in each component of DSP. For example, certain amount of power gets dissipated in DSP by the Dynamic Memory Access (DMA) module. On the other hand, the FPGA fabric has the logic design mapped onto it. The power dissipated

in this case is due to the activity level changes in the design and not any other module. Because we comparing the power consumption between DSP and FPGA implementations, we assume that power consumption in all the modules of the DSP except the CPU core is 0. This helps us to have a fair comparison between the two implementations. The values that are entered in the power estimation spreadsheet are:

1. Operating voltage
2. Desired case temperature
3. % CPU utilization

As stated in Chapter 3, XPower is used to calculate power dissipation for hardware implementation. It requires 3 files to calculate power dissipation – a) NGD file b) PCF file, and c) .VCD file. Out of these, the .VCD file is not generated during the *Implement* process in Xilinx ISE design flow. This file has to be generated using the Synopsys tool. We present the power analysis results for both types of implementations for both accelerators below. An operating voltage of 1.2 V and case temperature of 25°C were chosen for the purpose of analysis.

Matrix Multiplication

Table 5.9 shows the power analysis results for software implementation of 3×3 matrix multiplication. % Utilization is calculated for each implementation by multiplying the % time spent in doing useful task (% time spent for execution of the function) with the average % CPU utilization (60%). The power dissipation is a split of Baseline power and Activity power.

The power dissipation measurement of matrix multiplication was performed at an operating voltage of 1.2 V and an ambient temperature of 25°C. The total power dissipated was found to be **337.27** mW.

Table 5.9: DSP power analysis for Matrix multiplication

<i>Optimization Technique</i>	<i>% Utilization</i>	<i>Power Consumption (mW)</i>
Original code	19.46	654
Forward Store Optimization	21.13	659
Optimization <code>-o0</code> and F.S.	19.58	655
Optimization <code>-o1</code> and F.S.	13.099	637
Optimization <code>-o2</code> and F.S.	10.78	630
Unrolling innermost loop	6.617	619
Unrolling 2 inner loops	13.8264	639
Unrolling all the loops	9.056	625
Use of <code>_mpy</code>	5.97	617
Use of <i>reg</i> keyword	6.869	619

LU decomposition

The power dissipation analysis for LU decomposition was done in the same way as that of Matrix Multiplication. The amount of power dissipation for hardware implementation was found out to be **417.13** mW. The power dissipation for each software implementation of LU decomposition is shown in Table 5.10.

Table 5.10: DSP power analysis for LU decomposition

<i>Implementation Type</i>	<i>% Utilization</i>	<i>Power Consumption (mW)</i>
Original code	40.87	714
Optimization <code>-o0</code>	33.41	693
Optimization <code>-o1</code>	44.08	723
Optimization <code>-o2</code>	31.65	688
Use of <code>reg</code>	31.04	687
Use of <code>_mpy</code>	19.17	654
Use of <code>_fabsf</code>	19.01	653
Use of <code>_rcpsp</code>	18.15	651

Development Time

Development time can be measured in terms of the time required to perform the software and hardware implementations. An accurate comparison between the two types of implementations can be done by taking into account a) Number of man-hours required to perform the implementations b) Number of lines of code written to perform each implementation. It is difficult to measure the exact number of man-hours

required to perform the implementations. In most practical cases, an approximate comparison can be made by comparing the number of lines of code written to perform each implementation. In order to make an efficient comparison, every single line of code is written on a different line. Table 5.11 shows the number of lines of code written for each implementation. It can be seen from the table that for LU decomposition, the number of code lines written for hardware implementation is about 12 times the number of lines of code written for its software implementation. If a designer has equally good proficiency in both C and Verilog, the time required to perform hardware implementation can be 12 times the time required to perform software implementation. The process of hardware implementation does not stop at just writing the verilog code. The process of implementing the design on hardware also consumes significant amount of time. On the other hand, this is not the case for software implementation. It simply involves compiling the C code to generate binary code, which can be run on a DSP. Also, software debugging is much more easier and faster than hardware debugging. All the above mentioned facts show that software implementation of an algorithm is easier and faster than its hardware implementation.

Table 5.11: Code length comparison

Algorithm	Matrix Multiplication	LU decomposition
Software(C)	28	78
Hardware(Verilog)	83	932

5.2.6 Step 6: Analyze the results

This is the most critical part of the entire process. It deals with analysis of the results obtained from hardware and software implementations of the accelerator components. We will analyze the performance of both accelerators one after the other.

The section is divided into two parts. First, we analyze the results obtained for matrix multiplication, and next we analyze the results obtained for LU decomposition.

Matrix Multiplication

The comparison of Execution Time and Power Dissipation for software and hardware implementations can be seen in Figure 5.15 and Figure 5.16 respectively. The ideal requirements of an implementation are

- It should have the least execution time
- It should have minimum power dissipation

The best implementation can be found out by plotting a graph for all implementations with Execution Time on the x-axis and Power Dissipation on the y-axis. The graph can be seen in Figure 5.17.

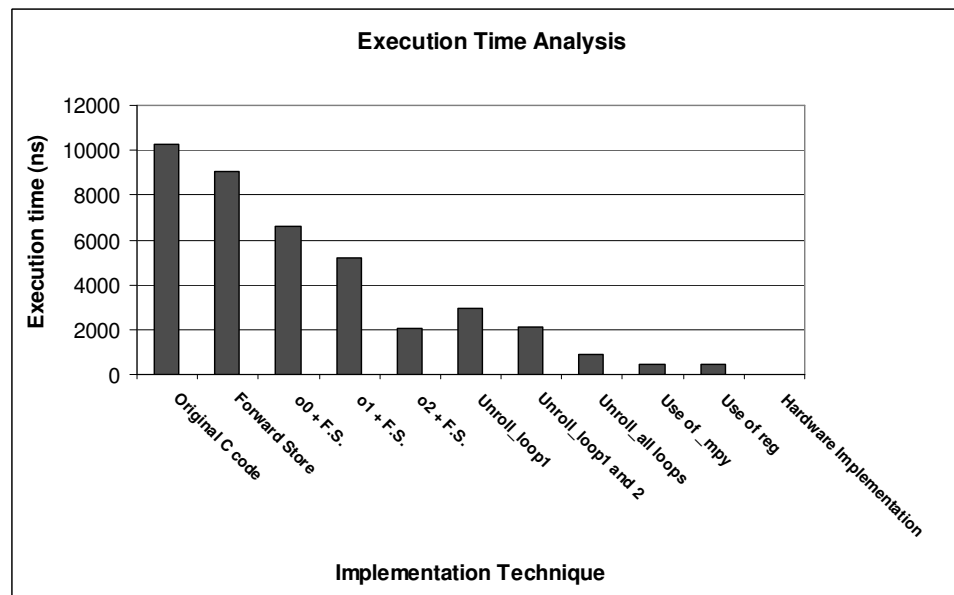


Figure 5.15: Execution Time analysis for Matrix Multiplication

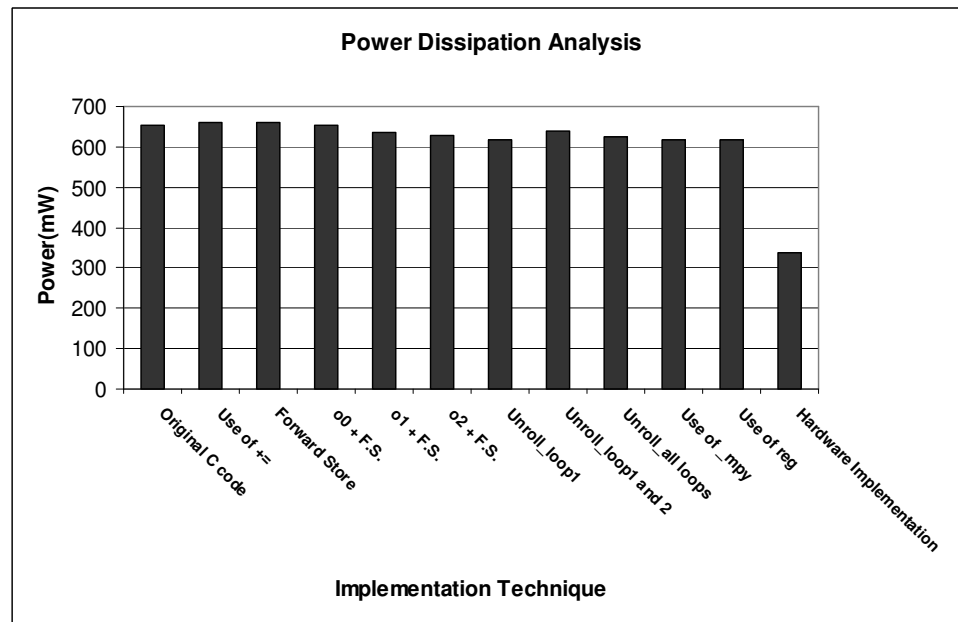


Figure 5.16: Power Dissipation analysis for Matrix Multiplication

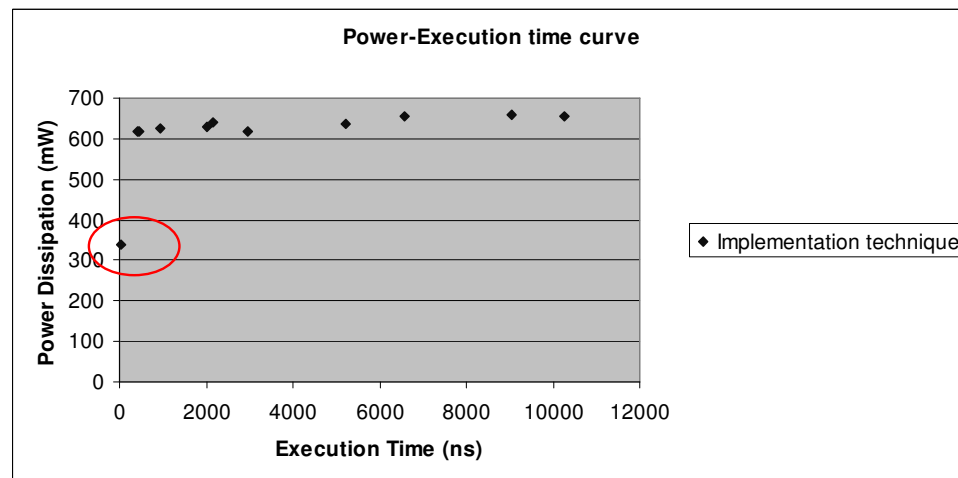


Figure 5.17: Execution Time – Power Dissipation curve

The point which represents minimum Power Dissipation and minimum Execution Time is the one which is closest to the origin. It has been highlighted in Figure 5.17. The implementation corresponding to the highlighted point is the best one in terms of Power Dissipation and Execution Time. It can be seen from Figures 5.15 and 5.16 that this point corresponds to hardware implementation of matrix multiplication. Both Execution Time and Power Dissipation of hardware implementation of matrix multiplication are better than the Execution Times and Power Dissipations of all software implementations. This shows that hardware implementation of matrix multiplication is efficient than its software implementation. The following calculations show the comparison between hardware and software implementations:

A) The best software execution time = 417.77 ns

Hardware execution time = 23.11 ns

Hence, the software execution time is 18 times the hardware execution time. In other words, the hardware implementation of matrix multiplication is $(417.77 - 23.11)/(417.77) = 95\%$ faster than its software implementation.

B) The least power dissipation for software implementation = 617 mW

Power Dissipation for hardware implementation = 337.27 mW

Hence, the power dissipation for software implementation is 1.8 times the power dissipation for hardware implementation. In other words, the hardware implementation of matrix multiplication is $(617 - 337.27)/(617) = 45\%$ more power efficient than its software implementation.

The above calculations show that hardware implementation of matrix multiplication is better than its software implementation. This does not necessarily mean that matrix multiplication should be implemented on FPGA. We need to take into consideration the communication overhead between DSP and FPGA before taking the final decision. The design can be implemented on FPGA if the summation of the time required to execute the design on FPGA and the time required for communi-

cation overhead is less than the time required to implement the design on DSP. In order to have an estimate of the communication overhead between DSP and FPGA, we take example of the P25M DSP/FPGA Integrated PCI card used for real time signal processing and control applications. It has the TI TMS320C6713 DSP connected to the Xilinx FPGA through a link which operates at a frequency of 75 MHz. The results obtained using this example are specific to this board. The results are highly dependent upon the speed of the link between DSP and FPGA. We calculate the time required for communication between DSP and FPGA as shown below:

For a 3×3 matrix multiplication, 18 32-bit numbers have to be sent from the DSP to the FPGA. If one data value is sent in one clock cycle, the time required to transmit 18 values can be calculated as shown below:

$$\text{Overhead time} = \frac{18}{75\text{MHz}} = 240\text{ns}$$

After the matrix multiplication operation is completed in the FPGA, the resultant matrix has to be transmitted back to the DSP. The resultant matrix has 9 elements which will take 120 ns to get transmitted to the DSP. So, the total overhead time = $240 + 120 = 360$ ns. Hence the summation of the time required for hardware implementation and the time required for communication is $23.11 + 360 = 383.11$ ns. This is still less than the time required to perform software implementation of matrix multiplication which is 417.77 ns. It can be observed that there is no significant performance (just 8%) improvement obtained by implementing the matrix multiplication operation on FPGA. Also, when data is transmitted between DSP and FPGA, significant amount of power dissipation takes place.

Moreover, the matrix multiplication operation needs to be performed 11 times in the entire ADC algorithm. This means that if this operation is implemented on FPGA, a significant amount of time is spent in communication overhead. The time required for hardware implementation = $11 \times 23.11 = 254.21$ ns, and the time required for communication overhead = $360 \times 11 = 3960$ ns. This shows that the

amount of time spent in communication overhead is 16 times the amount of time spent in the actual implementation. Although the power dissipation in FPGA is less than the power dissipation in DSP, every time data is transferred between the two, some amount of power dissipation will be there. All the above facts lead to a conclusion that the matrix multiplication operation should be implemented in software (DSP) and not in hardware (FPGA).

LU decomposition

The results obtained for LU decomposition have been analyzed using the same method as the one used for matrix multiplication. The comparison of Execution Time and Power Dissipation for software and hardware implementations of LU decomposition can be seen in Figure 5.18 and Figure 5.19 respectively. The best implementation in this case is also found out by plotting a graph for all implementations with Execution Time on the x-axis and Power Dissipation on the y-axis. The graph can be seen in Figure 5.20.

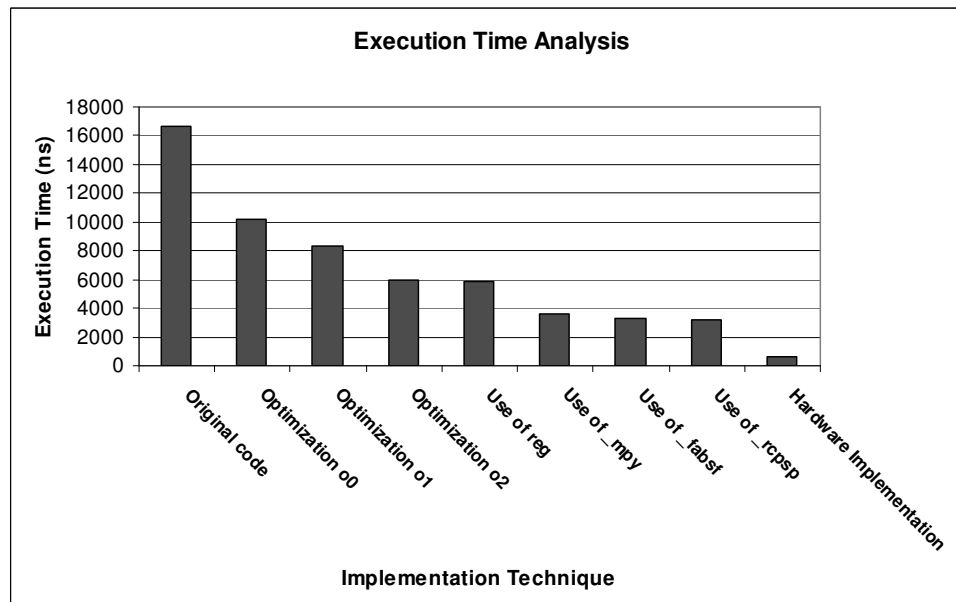


Figure 5.18: Execution Time analysis for LU decomposition

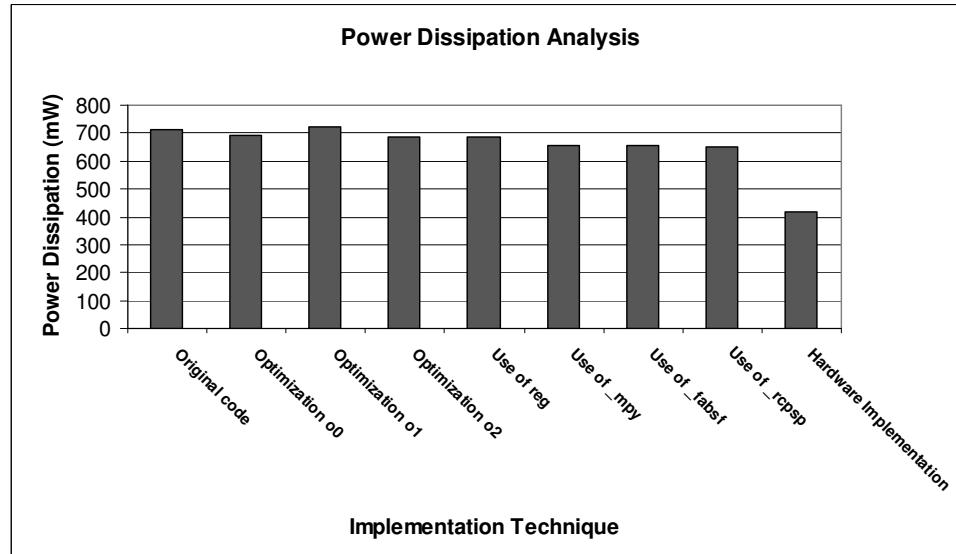


Figure 5.19: Power Dissipation analysis for LU decomposition

As in the case of matrix multiplication, the point which is closest to the origin in Figure 5.20 is the one with least execution time and least power dissipation. This is the point which represents hardware implementation of LU decomposition. It can be seen in Figures 5.18 and 5.19 that the hardware implementation of LU decomposition has lower execution time and lower power dissipation than all the software implementations. The following calculations show the comparison between hardware and software implementations:

A) The best software execution time = 3186.67 ns

Hardware execution time = 658.97 ns

Hence, the software execution time is 4.8 times the hardware execution time. In other words, the hardware implementation of LU decomposition is $(3186.67 - 658.97)/(3186.67) = 80\%$ faster than its software implementation.

B) The least power dissipation for software implementation = 651 mW

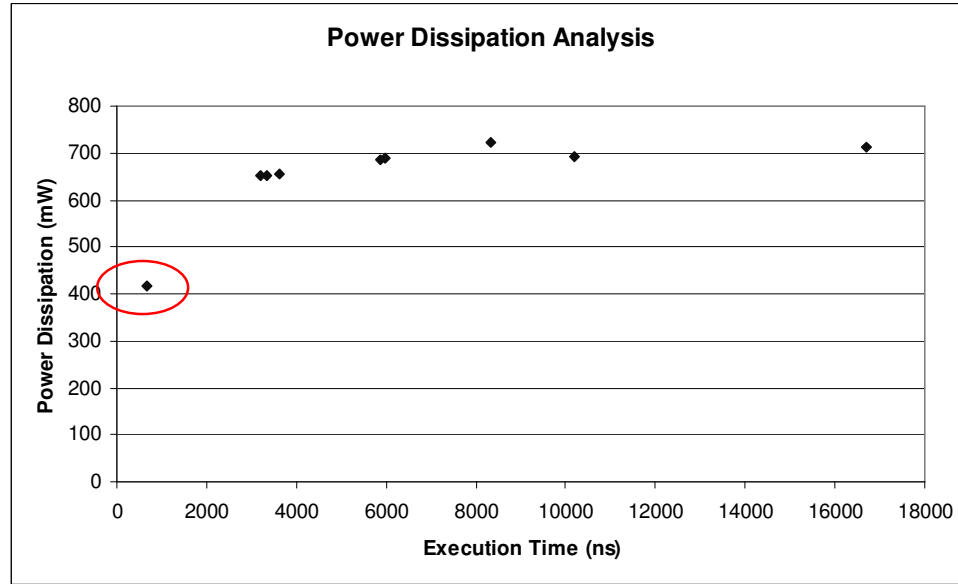


Figure 5.20: Execution Time – Power Dissipation curve

Power Dissipation for hardware implementation = 417.13 mW

Hence, the power dissipation for software implementation is 1.5 times the power dissipation for hardware implementation. In other words, the hardware implementation of LU decomposition is $(651 - 417.13)/(651) = 36\%$ more power efficient than its software implementation.

The above calculations show that hardware implementation of LU decomposition is better than its software implementation. But we need to consider the communication overhead between DSP and FPGA. We perform the same analysis as performed in the case of matrix multiplication.

For LU decomposition of a 3×3 matrix, 9 32-bit numbers have to be sent from the DSP to the FPGA. If one data value is sent in one clock cycle, the time required to transmit 9 values can be calculated as shown below:

$$\text{Overhead time} = \frac{9}{75\text{MHz}} = 120\text{ns}$$

After the decomposition process is completed in the FPGA, the resultant matrix has to be transmitted back to the DSP. The resultant matrix has 9 elements which will take 120 ns to get transmitted to the DSP. So, the total overhead time = $120 + 120 = 240$ ns. Hence the summation of the time required for hardware implementation and the time required for communication is $658.97 + 240 = 898.97$ ns. This is still less than the time required to perform software implementation of LU decomposition which is 3186.67 ns. It can be observed that there is still a significant performance (72%) improvement obtained by implementing the LU decomposition process on FPGA.

Moreover, the LU decomposition process needs to be performed only 1 time in the entire ADC algorithm. So, if it is implemented on FPGA, there can be a significant improvement in the performance as well as power dissipation. As seen earlier, the power dissipation in the case of FPGA implementation is 36 % less than the power dissipation in the case of DSP implementation. All the facts presented above lead to a conclusion that the LU decomposition of 3×3 for this algorithm should be implemented in hardware.

5.3 Conclusion

In this chapter, we have analyzed the Attitude Determination and Control algorithm using the proposed methodology. The complexity analysis of the algorithm identified Matrix Multiplication and LU decomposition as the most computationally complex blocks of the algorithm. As stated by the methodology, we performed software and hardware implementations of these blocks. The software implementations were optimized using various techniques to improve the performance of the C codes. Next, we measured Execution Times, Power Dissipations and Development Times for all the implementations. Using these metrics, we compared the hardware and software implementations. We analyzed the results by performing several numerical calculations. After analyzing the results, we came to a conclusion that Matrix

Multiplication should be implemented in software and the LU decomposition should be implemented in hardware. Hence, using the proposed methodology, we have transformed the Simulink model from a Platform Independent Model (PIM) to a Platform Specific Model (PSM).

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis has presented a methodology for analyzing complex algorithms for space based applications. There is a variety of hardware architectures available commercially in which the algorithms can be mapped. Also, there exist different design tools suiting every architecture which can be used to perform the implementation of these complex algorithms on appropriate hardware architectures. A number of methodologies have been proposed for implementing complex algorithms for wide variety of applications. But there does not exist a methodology which provides a step by step process of analyzing algorithms for space based applications. An important requirement of any design to be used for a space based application is that it should have low power dissipation. The methodology presented in this thesis takes this factor into consideration while making the implementation decisions.

The methodology proposed in this thesis provides a step by step process of analyzing an algorithm for converting its Platform Independent Model (PIM) into a Platform Specific Model (PSM). It assumes that the algorithm has been designed in

Simulink from Mathworks, which is the design tool of choice for space engineers. The methodology starts with selecting the candidate platforms on which the algorithm can be implemented. After that, it describes the process of complexity analysis of the algorithm to find out the most computationally complex blocks of the model. The next step deals with software and hardware implementations of the chosen blocks (accelerators). The software implementation flow is specific to TI DSPs and the hardware implementation flow is specific to Xilinx FPGAs. After performing these implementations, the next step is to measure the metrics of comparison like Execution Time, Power Dissipation and Development Time for these implementations. The last step of the methodology explains the process of analyzing the results and taking decision regarding the choice of platform (DSP or FPGA) on which the accelerators should be implemented. This methodology leads to a conclusion whether the entire algorithm can be implemented on DSP or the computationally complex parts of the algorithm can be implemented on FPGA.

This proposed methodology is validated by applying it to the Attitude Determination and Control algorithm designed for attitude control of small satellites. DSP and FPGA are chosen as the candidate platforms for implementing the algorithm. The computationally complex blocks of the Simulink model of the algorithm are determined using Time Complexity Analysis. This analysis shows that Matrix Multiplication and LU decomposition are the most complex blocks of the algorithm. Software implementation of these blocks is performed for TI TMS320C6713 DSP and hardware implementation is performed for Xilinx Virtex-4 FPGA. The codes written for software implementation are optimized using a number of optimization techniques. After performing the implementations, metrics like Execution Time, Power Dissipation and Development Time are measured for both the accelerators. It is found that hardware implementations of both Matrix Multiplication and LU decomposition are faster and more power efficient than their corresponding software implementations. Although hardware implementation proves out to be more efficient than software implementation, it is found that matrix multiplication should be implemented on DSP (software) and LU decomposition should be implemented on FPGA (hardware). This is deduced

from the results obtained after taking the cost of overhead communication between DSP and FPGA into consideration shown in Section 5.2.6. Table 6.1 summarises the results obtained for both Matrix Multiplication and LU decomposition.

Table 6.1: Performance Improvement for Matrix Multiplication and LU decomposition

<i>Performance Metric</i>	<i>Matrix Multiplication</i>	<i>LU Decomposition</i>
DSP Execution Time	417.77 ns	3186.67 ns
FPGA Execution Time	23.11 ns	658.97 ns
Overhead Cost	320 ns	240 ns
% Improvement in speed	8 %	72 %
DSP Power Dissipation	617 mW	651 mW
FPGA Power Dissipation	337.27 mW	417.13 mW
% Improvement in Power Dissipation	45 %	36 %

Apart from providing a directed flow for analyzing algorithms, the thesis also discusses the limitations of some commercially available tools like Real Time Workshop by TI and System Generator by Xilinx. The limitations of Real Time Workshop are a) The code generated from Simulink model is not an efficient code b) The code has poor readability, and c) The code cannot be compiled by all commercially available compilers. The case study of an FIR filter shows the validity of the above mentioned points. During the course of our research, we have gained knowledge about a number of commercially available tools like Xilinx ISE, TI Code Composer Studio, Simulink, Real Time Workshop, Link for Code Composer Studio, Xilinx System Generator and Xilinx XPower. Based on our experience, we have given information about the use of

these tools for different applications. Using this information, a design engineer can expedite the design process without spending time in tool exploration. We feel that the methodology proposed in this thesis is of value to designers involved in space based projects, as it provides a directed flow for analyzing complex space based algorithms in a short amount of time.

6.2 Future Work

The methodology proposed in this thesis has been used to analyze the Attitude Determination and Control algorithm designed for attitude control of small satellites. The results of the analysis of the algorithm should be used to actually perform the implementation of the algorithm on a board which has both DSP and FPGA on it. The board to be selected for performing the implementation should be available commercially and should meet the requirements of the platform in which it has to be reside. We are planning to use the Cubesat kit as the platform for accomplishing the small satellite operation. The board selected for implementing the algorithm should have dimensions and bus interface matching the specifications of the Cubesat kit. One such board is the P25 DSP/FPGA PCI card with Analog I/O. It has 64-bit 66 MHz PCI bus which can be interfaced with the PCI bus of the Cubesat kit. Also, it has A/D converters which can be used to convert the analog signals coming from the sensors to digital form and transmit them to the DSP. Also, this board has support for the TI and Xilinx tools used in the proposed methodology. This board seems to be an ideal match for implementing the ADC algorithm.

There are number of similar algorithms being designed at the AMAS laboratory at University of Florida, Gainesville. One of them is the algorithm which provides translational motion to the SGCMG cluster shown in Figure 5.1. All these algorithms will be analyzed and implemented using the methodology proposed in this thesis. One of our long term goals is to develop a tool which automates the steps described in this methodology.

Bibliography

- [1] F. Leve, “Design of 3-DOF testbed for micro-satellite autonomous operations,” *45th AIAA Aerospace Sciences Meeting and Exhibit, 8 - 11 January 2007, Reno, Nevada.*
- [2] W. R. Davis, “A hierarchical, automated design environment for low-power, high-throughput digital signal processing ics,” *PhD Dissertation, University of California, Berkeley, January 2002.*
- [3] D. Hercog, M. CurkoviE, G. Edelbaher, and E. Urlep, “Programming of the DSP2 board with the Matlab/Simulink,” *The IEEE International Conference on Industrial Technology (ICIT 2003), Maribor, Slovenia.*
- [4] D. D. Gajaski and L. Ramachandran, “Introduction to high level synthesis,” *IEEE Design and Test of Computers, vol. 11, no.4, pp.44-54, 1994.*
- [5] M. Todd and V. Amanda, “A platform approach to small satellite design,” *54th International Astronautical Congress of the International Astronautical Federation, the International Academy of Astronautics, and the International Institute of Space Law, Bremen, , Sep. 29-3, 2003.*
- [6] “Classification of satellites on the basis of their wet mass.” <http://centaur.sstl.co.uk/SSH/sshpclassify.html>.
- [7] “An example of a pico-satellite.” <http://www.cubesatkit.com>.
- [8] “Space and systems development laboratory.” <http://ssdl.stanford.edu>.

- [9] G. Angelino, "Approximate method for plug nozzle design," *AIAA Journal*, Vol. 2, No. 10, pp.1834-1835.
- [10] L. Adams, "Choosing the right architecture for real time signal processing designs," *Texas Instruments - White Paper SPRA879 - November 2002*.
- [11] "Object management group, mda guide version 1.0.1." <http://www.omg.org/docs/omg/03-06-01.pdf>.
- [12] F. Bolandl, A. Chureau, C. Thibeaup, Y. Savaria, F. Gagnod, and Z. Zilic, "An efficient methodology for design and verification of software defined radio," *Circuits and Systems, 2004. NEWCAS 2004*.
- [13] "True DSP Synthesis - The Birth of a new design methodology - synplicity." Synplicity, Inc.White Paper.
- [14] J. Brown and M. Epalza, "Automatically identifying and creating accelerators directly from c code," *Xcell Journal - Third Quarter 2006* pp 58-60.
- [15] "Xilinx platform studio." http://www.xilinx.com/ise/embedded/edk_pstudio.htm.
- [16] A. Baganne, J. L. Pillipe, and E. Martin, "A co-design methodology for telecommunication systems: A case study of an acoustic echo canceller," *VLSI Signal Processing*, vol. 22, pp. 21-29, 1999.
- [17] Martin, O. Sentieys, H. Dubois, and J. Philippe, "Gaut: An architectural synthesis tool for dedicated signal processors," *Proc. of EURO-DAC*, 1993.
- [18] Y. Sorel, "Massively parallel computing systems with real time constraints: The algorithm architecture adequation," *Proc.Of Massively Parallel Computing Systems Conference, 1994, Italy*, p.p. 44-53, 1994.
- [19] M. A. Shanblatt and B. Foulds, "A simulink-to-fpga implementation tool for enhanced design flow," *Proceedings of the 2005 IEEE International Conference on Microelectronic Systems Education (MSE-05)*.

- [20] C. Rowen and S. Leibson, "Flexible architectures for engineering successful SOC's," *Design Automation Conference, 2004. Proceedings. 41st*, pp. 692–697, June 2004.
- [21] "The international technology roadmap for semiconductors - 2003 edition."
<http://public.itrs.net>.
- [22] T. Gemmeke, M. Gansen, H. J. Stockmanns, and T. G. Noll, "Design Optimization of Low Power High Performance DSP Building Blocks," *IEEE Journal of Solid-State Circuits*, Vol. 39, No. 7, July 2004.
- [23] G. Goossens, J. V. Praet, D. Lanneer, W. Geurts, A. Kifli, C. Liem, and P. G. Paulin, "Embedded software in real-time signal processing systems: Design technologies," *Proceedings Of the IEEE*, VOL. 85, NO. 3, MARCH 1997.
- [24] E. A. Lee, "Programmable DSP Architectures Part I," *IEEE Acoustics, Speech, and Signal Processing Magazine*, October, 1988, p.p. 4-19.
- [25] E. J. Tan and W. B. Heinzelman, "DSP Architectures: Past, Present and Future," *Department of Electrical and Computer Engineering, University of Rochester, Rochester, NY 14627, White Paper*.
- [26] B. Zahiri, "Structured ASICs: opportunities and challenges," *21st International Conference on Computer Design* pp. 404-409, October 2003.
- [27] J. Rabaey, "Reconfigurable processing: the solution to low-power programmable DSP," *IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, Munich, Germany, Apr. 1997. pp. 275-278.
- [28] S. Mathworks Inc., *Dynamic System Simulation for Matlab*. Natick, Massachusetts, 1998.
- [29] A. Sangswang, G. Rost, and C. Nwankpa, "A modular simulink-based controlled three phase switch mode inverter," *IEEE Power Engineering Society Summer Meeting, 2000* p.p. 2101-2106 vol. 4, Seattle, WA, USA.

- [30] S. Mathworks Inc., *Real-Time Workshop User's Guide*. Natick, Massachusetts, 1998.
- [31] *Xilinx System Generator v2.1 for Simulink, User Guide*.
- [32] A. Sharma and C. Ravikumar, "Efficient Implementation of ADPCM Codec," *Thirteenth International Conference on VLSI Design, 2000* p.p. 456-461.
- [33] "Xilinx ISE Flow." <http://www.xilinx.com/ise>.
- [34] "Texas instruments code composer studio." <http://focus.ti.com/docs/toolsw/folders/print/ccstudio.html>.
- [35] K. Ulrich and S. Eppinger, *Product Design and Development*. McGraw-Hill, 2003.
- [36] M. Ros and P. Sutton, "Compiler optimization and ordering effects on vliw code compression," *School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, Australia 4072*.
- [37] "Forward store optimization." <http://www.nullstone.com>.
- [38] "Xilinx ise." http://www.xilinx.com/ise/logic_design_prod/webpack.htm.
- [39] S. Kuo and W. Gan, "Digital signal processors : Architectures, implementations and applications," *Pearson Education, Inc, Upper Saddle River, New Jersey 07458: Pearson Prentice Hall, 2005*.
- [40] "Fixed-Point Arithmetic - part-II," *Real Time Digital Signal Processing EEE 404/591 Lecture Notes, A.S. University, 2007*.
- [41] Texas Instruments TMS320C6711D, C6712D, C6713B Power Consumption Summary. Application Report. SPRA889A - August 2005.
- [42] XPower Tutorial - FFPGA design. XPower (v1.3) July 15, 2000.

- [43] R. Banning, M. Roesch, and A. Morgan, “Improved embedded flight system design process using integrated system design/code generation tools,” *Honeywell Inc. Defense Avionics Systems Albuquerque, NM 87113-2227*.
- [44] Texas Instruments TMS320C6000 Optimizing C Compiler Tutorial (SPRU425A), August 2002.
- [45] B. Varnagiryte, A. Zemelis, O. Olsen, P. Koch, O. Wolf, and E. Kazanavicius, “A practical approach to DSP code optimization using compiler/architecture,” *ISSN 1392-2114 ULTRAGARSAS, Nr.2(43). 2002. White Paper*.
- [46] B. Wie, D. Bailey, and C. Heiberg, “Singularity robust steering logic for redundant single gimbal control momentum gyros,” *AIAA Guidance, Navigation, and Control Conference and Exhibit, Denver, CO, Aug. 14-17, 2000*.
- [47] “Time complexity of square root operation.”
<http://numbers.computation.free.fr/Constants/Algorithms/inverse.html>.
- [48] Z. Nikoli, H. T. Nguyen, and G. Frantz, “Design and Implementation of Numerical Linear Algebra Algorithms on Fixed Point DSPs,” *Texas Instruments Inc. 12203 SW Freeway, MS722, Stafford, TX 77477*.
- [49] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C - The Art of Scientific Computing*. Cambridge University of Press, 1988.
- [50] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2001.