An Application Framework for a Power-Aware Processor Architecture

Anup Mandlekar

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Mark T. Jones, Chair
Thomas L. Martin, Co-Chair
Paul E. Plassmann

August 7, 2012
Blacksburg, Virginia

Keywords: Model Driven Engineering, Simulink,
Dataflow Architecture, Low Power Flash Memory Cells

An Application Framework for a Power-Aware Processor Architecture

Anup Mandlekar

(ABSTRACT)

The instruction-set based general purpose processors are not energy-efficient for event-driven applications. The E-textiles group at Virginia Tech proposed a novel data-flow processor architecture design to bridge the gap between event-driven applications and the target architecture. The architecture, although promising in terms of performance and energy-efficiency, was explored for limited number of applications. This thesis presents a model-driven approach for the design of an application framework, facilitating rapid development of software applications to test the architecture performance. The application framework is integrated with the prior automation framework bringing software applications at the right level of abstraction. The processor architecture design is made flexible and scalable, making it suitable for a wide range of applications. Additionally, an embedded flash memory based architecture design for reduction in the static power consumption is proposed. This thesis estimates significant reduction in overall power consumption with the incorporation of flash memory.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0834490.

Acknowledgements

First and foremost, I would like to express my gratitude towards my advisor Dr. Mark Jones as it has been a great learning experience working with him. The constructive suggestions and criticisms offered by him have not only helped in shaping up this research but have also helped me grow as an individual, both professionally and personally. I truly appreciate all his support and guidance.

I would like to thank Dr. Thomas Martin for his valuable and insightful suggestions throughout the course of my research work. His opinions and reviews about the work have always been very helpful and appreciated. I would also like to express my gratitude to Dr. Paul Plassmann for serving in my thesis committee.

I would like to thank Mr. Sarosh Malayattil, my research partner, and Mr. Jason Forsyth for their helpful suggestions. Also, I would like to specially mention my friends Sarvesh, Supratik, Apoorv, Dhaval, Kaushik, Pallavi, and Ankit for always being a great support.

Last but not the least, the constant support I have received from my parents and my sister has been decisive, and I thank them for it.

Anup Mandlekar

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last few decades, the field of electronic textiles (e-textiles) has emerged as an enabling technology for wearable and pervasive computing [1]. E-textiles are traditional fabrics that enable computing, digital components, and electronics to be embedded in them in an unobtrusive manner [2]. Because e-textiles provide an excellent platform for integrated computing, sensing, and communication capabilities in durable fabrics, they have a wide range of applications in the real world including medical, construction, and military applications [1].

A seamless integration of electronics in textiles demands a close coupling of sensing and computing components to the physical structure of the fabric [3]. This imposes power and energy constraints on the computing components [4], [5]. General purpose processors, such as PIC® micro-controllers, currently used as computing nodes woven into the textiles, do not offer an efficient solution as there is an inherent mismatch between their control-flow based processor architecture and the event-driven nature of e-textiles applications [6]. To address this issue, the VT E-Textiles Group [7] designed and developed a power-aware data-flow architecture which aims to be an energy and performance efficient target for such applications.

The processor architecture design, although promising in terms of performance and energy efficiency, was tested for a limited number of applications [6][8]. This thesis takes a Model-Driven-Engineering (MDE) approach for the development of an application framework which aims for a rapid development of the software applications for the new architecture.

1.1 Motivation

The design and development of a novel processor architecture and an associated software framework to bridge the gap between the event-driven applications and the target processor architecture is presented in [6][8]. The processor architecture is implemented using data-flow principles to suit the event-driven nature of the applications. Additionally, an automation framework is designed to achieve rapid prototyping of experimental processor architectures, automatic generation of configurations, and automatic validation of the implementations. The processor design aided by the software framework is shown to be a good target for the MDE approach as compared to general purpose micro-controllers [6]. The implemented processor architecture design is also shown to be an energy efficient target as compared to the PIC12F675 general purpose micro-controller [6].

However, the previous implementation had certain design limitations. The lack of application development structure in the previous framework made application specification a complex process. It demanded significant manual efforts in application programming. It was important to bring the previous framework to the right level of abstraction to simplify the application design process. The previous version of the architecture was also limited by its packet structure, restricting the number of modules in the architecture to *eight*. This restricted the class of applications that can be executed on this processor architecture. Ad-

ditionally, a significant static power consumption was observed during the idle states of the application execution in spite of using a power-aware design approach [8].

This thesis addresses these limitations through an MDE-based application framework, a modified communication packet and template structure to enhance the flexibility of the architecture, and the incorporation of flash memory into the architecture to achieve significant leakage power savings.

1.2 Contributions

This thesis makes three important contributions towards the design of an application programming interface for power-aware data-flow processor architecture. First, a Simulink [9] based application modeling framework is designed and implemented, providing a right level of abstraction for the applications. It builds upon the existing framework and allows a rapid application development in the Simulink graphical user interface, facilitating a high application quality and a simplified validation. Second, a flexible communication packet structure and template for the functional units of the architecture is developed, expanding the architecture exploration space. It removes the prior limitations on the configuration and design decisions making it scalable and adaptable for the desired range of event-driven applications. Third, flash memory based architecture is proposed to achieve a significant reduction in the power consumption. Flash memory has a capability to retain data without a power supply, which allows portions of the architecture to be switched off during the idle states, resulting in significant savings in the total power and energy consumptions.

The quality of the application modeling framework is demonstrated by using it to successfully generate application configurations from an application suite developed in Simulink. The flexibility of the redesigned architecture is validated by executing a diverse range of

applications. The power efficiency of the flash memory based architecture is demonstrated by comparing the power savings with respect to the non-flash based architecture.

1.3 Document Overview

The rest of this thesis is organized as follows. Chapter 2 discusses the research work related to this thesis. Chapter 3 describes the implementation of an MDE-based application modeling framework for the application suite generation. Chapter 4 discusses the design and implementation of a flexible template structure of the architecture making it suitable for a range of applications. It also describes the incorporation of low power flash memory into the design. Chapter 5 presents the experimental results demonstrating the quality of the thesis contributions. Chapter 6 concludes the thesis with a summary of contributions and a description of future work.

Chapter 2

Related Research

This chapter discusses research related to this thesis. A brief introduction of Model-Driven Engineering and related research in model-driven design environments for application development is given in Section 2.1. Section 2.2 discusses several architectural techniques for low power consumption and energy efficiency, including embedded flash memory.

2.1 Model-Driven Engineering Overview

This thesis incorporates Model-Driven Engineering (MDE) principles for developing an application programming interface to test the processor architecture. Section 2.1.1 gives a brief introduction of MDE. Section 2.1.2 describes popular MDE-based products and tools that offer a development environment with automated validation. Section 2.1.3 discusses the related research work in model-driven design environments for application development.

2.1.1 Introduction to Model-Driven Engineering

Model-driven engineering (MDE) is a software development paradigm which focuses on creating application specific abstract representations, instead of the underlying computing environments. This approach eliminates the need to consider the computing platform complexities, allowing the developers to program according to their design requirements [10]. MDE methodology has become very popular as it elevates software applications development to a higher level than possible with third-generation programming languages [10]. The third generation programming languages (3GL), such as FORTRAN, C, C++, C#, raised the level of abstraction, however, the dependency on the computing environment was not completely eliminated. This inability of the 3GL coupled with the capability of the MDE approach to abstract the platform complexity and precisely express the domain concepts, makes MDE an effective methodology [11].

Model-Driven Engineering combines two major functionalities, *domain specific modeling languages* (DSML) and *transformation engines and generators* [10]. Developers make use of DSMLs to build applications using models that express logic of the design intent without describing its control flow explicitly using program statements. *Transformation engines and generators* analyze certain aspects of these models and generate source code or simulation inputs or both, depending upon the framework requirement [10]. A few examples of the MDE-based tools are discussed in the next section.

2.1.2 Model-Driven Engineering Tools

There are several commercially successful MDE products that are being used in various industries such as Aeronautics, Automotive industry, Cyber-Physical Systems, and Industrial Automation. A brief description of some of the popular MDE products is provided

in this section and the reason behind the selection of Simulink as the application modeling environment for the thesis is discussed.

Simulink [9] offers a comprehensive model-based design environment for the development and simulation of dynamic and embedded systems. Developers can utilize the extensive libraries of predefined blocks as well as user-defined subsystems to visually create an application in its graphical environment. It also offers automatic code generation with *Simulink Embedded Code Generation* [12] from the designed *Simulink* block diagram for the simulation and validation of the system behavior.

Ptolemy II [13] is another model-based framework aimed at providing a methodology for defining and producing embedded software applications. Like *Simulink*, it provides a set of models of computation with which the user can construct a system in its graphical environment. It supports several models of computation facilitating simplified integration of complex heterogeneous systems, unlike *Simulink*. It is used for modeling of communication networks, circuit designs, and for design assistance for hardware/software co-design. However, it has a limited model library, lacks periodic discrete-time support, and does not offer automated code generation.

Reactis [14] provides model-based solution to automatically test and validate the model generated for the application developed in *Simulink*. It offers model-based testing, debugging, and validation for *Simulink* and *Stateflow* [15] models. *Reactis Tester* automatically generates test suites from *Simulink* and *Stateflow* models [14]. These test suites are primarily used to verify whether the generated code conforms to the user-specified requirements. However, the generated code is sometimes inefficient in terms of energy and performance because of the mismatch between the computational model used and the target architecture.

There are several MDE-based tools languages such as *Unified Modeling Language* (UML) [16]

which uses graphic notations to create visual models of the object-oriented systems. Being a general-purpose language, it is not capable of providing features which are appropriate for expressing the domain-specific problems. Several variations have been proposed to make UML suitable for certain domain-specific situations. For eg, *MARTE* [17] adds capabilities to UML for model-driven development of Real Time and Embedded Systems. The *ALDERIS* [18] language has both a visual and textual syntax to represent distributed real-time embedded systems. *PeaCE* [19], an extension of *Ptolemy II* project, offers a model-driven specification and automatic code generation for software and hardware co-design implementations.

Simulink is consciously chosen to be the functional modeling language for the development of the application framework in this thesis. This is because, its extensive libraries of predefined blocks and user-defined subsystem provide the right level of abstraction for the application representation. Also, it is integrated with Matlab, a functionally specialized and popular tool with a powerful math library, to simulate the behavior of the model. However, its code generation capabilities are not suitable for the program execution on the designed dataflow architecture. Therefore, the *Simulink* environment is interfaced with a custom built application code generation framework to extract desired application graph from the model.

2.1.3 MDE Design Environments for Application Development

Multiple efforts have appeared over the years supporting model-based application development. This section discusses several recent research efforts that demonstrate the success of model-based design for application development and automated validation.

A novel model-based programming environment of embedded software applications for Multi-Processor System on a Chip is proposed in [20]. The programming framework allows generation of diverse models for the architecture specification by modeling the software in MDE-

based languages such as UML and PeaCE. The specification is converted to a fixed common intermediate code (CIC) format, which consists of necessary hardware information such as address mapping of memory segments, power and performance constraints. Thus, it separates modeling of the software from the target architecture [20]. It also provides multi-phase debugging capabilities: at the modeling stage, at the code generation stage, and at the simulation stage [20].

A new approach, termed as SysWeaver, is offered in [21] for the model-based embedded development. SysWeaver is a model-based design tool that includes a flexible code generation scheme for distributed real-time systems that can be easily tailored to a wide range of target platforms [21]. In this approach, the functional aspects of the system are specified in Simulink and translated into a SysWeaver model to be enhanced with timing information, the target hardware model and its communication dependencies. To be able to automate the code generation for the entire system, the Simulink models are imported and converted to SysWeaver functional models, which involves mapping a Simulink data-flow model into the SysWeaver control flow model. The integrated chain offers the developer the advantages of both tools including analysis, simulation, verification, and complete code generation [21].

A model-based framework for functional verification and performance estimation of chip multiprocessors (CMPs) is introduced in [22]. CMPs consist of several complex components such as programmable processors, custom logic blocks, and memories, all of which are connected together via an interconnection network. The model-based (CARTA) framework utilizes ALDERIS [18] DSML for the formal modeling of the CMPs. The key properties of the CMP design, such as its structure, interconnection network, and behavior are specified using ALDERIS. By switching to an abstract representation of the design, significant reduction in the overall design time with negligible accuracy loss is achieved as compared to a sequential design process [22].

The MDE principles are shown to have great significance in System on Chip (SoC) design, which requires to master a lot of different abstraction levels, different simulation techniques, and different synthesis tools [23]. The application and the target hardware architecture are described by different meta-models. Therefore, the reuse of hardware and software components becomes easier as they are graphically defined in a modeling environment such as UML. ModTransf [23], a simple transformation engine is built to generate a code for the validation of the hardware architecture functionality.

The application modeling framework built in this thesis partially follows the MDE paradigm offered in [21][20]. Simulink model is chosen as the high-level design abstraction for the application representation. Its simulation environment is used to generate the architecture validation artifacts. However, instead of utilizing automated code generation functionality of Simulink, the functional aspects of the application specified in Simulink are translated to an intermediate format termed as application model graph. The custom built code generation framework is used for this translation. This intermediate format is mapped on the data-flow architecture description to be able to automate the code generation. The integrated tool-chain offers automated simulation and verification of the proposed data-flow architecture.

2.2 Related Research in Power-efficient Processor Architectures

Apart from the model-driven application framework, this thesis concentrates upon the power-efficiency of the designed data-flow architecture. The related research in the architecture techniques for low power consumption in processor architectures is discussed in Section

2.2.1. An overview of the embedded flash memory and its application in power-critical field is provided in Section 2.2.2.

2.2.1 Architecture Techniques for Low Power Consumption

The designed data-flow architecture is specifically targeted for event-driven applications of sensor monitoring in e-textiles computing space [8]. As these systems are battery powered, the performance of the system processor nodes is required to be optimized for power efficiency. Therefore, it is important for the processor architecture to match the low power demands of these systems. This section presents an overview of research efforts in power-efficient architectures.

An ultra low power system architecture for sensor network applications is proposed in [24]. It replaces the instruction set-based functionality of a general-purpose micro-controller with an event-driven system consisting of coarse-grained modules. It provides an easily extensible system architecture that allows different sets of hardware components to be combined into a larger system, targeting a particular application. The event-driven nature of this architecture facilitates switching off any unused modules to reduce the total power consumption. The power-down process of the modules of the architecture is aided by the custom designed on-chip SRAM [24].

A power efficient processor for sensor applications, called the Phoenix Processor, is proposed in [25]. Like the ultra low power system architecture, it offers an event-driven CPU with compact ISA, a custom built low leakage memory cell, and an adaptive leakage management in the data memory. Additionally, the Phoenix processor utilizes a power gating approach using high threshold transistors. The processor is shown to be a power-efficient target for the battery powered sensor nodes [25].

The continuing decrease in the feature size of CMOS process technology also facilitates a denser and higher processor performance [26]. However, a dominance of leakage power over dynamic power in the smaller feature size process technology comes in the way of efficient processor performance [26]. Various leakage-aware design techniques are proposed for different process technologies. For instance, dynamic voltage scaling is a leakage power control mechanism effective at 130nm process [27]. It reduces the power consumption of processors when peak performance is unnecessary. However, the same technique is not effective for processes below 100nm.

The power-aware design proposed in [8] utilizes the power gating approach for the power management of the functional units suited for the process technology of 90nm and below. The power management principle implemented in this design is similar to the device-level management discussed in [24]. A voltage island phenomenon is utilized for reducing the power consumption [8] with the aid of Unified Power Format (UPF) [28]. These voltage islands are switched off when not in use. However, the modules of the architecture are not completely switched off, as they contain configuration information necessary for the correct functional behavior of the architecture. Therefore, a significant leakage power consumption is observed when the functional modules are idle. This thesis proposes the utilization of embedded flash memory cells to retain the configuration information in the idle states, facilitating complete power-down of the functional modules. A brief overview of embedded flash memory is provided in the next section.

2.2.2 Embedded Flash Memory Overview

The embedded flash memory can be defined as an electrically erasable non-volatile memory that is physically and electrically integrated on a monolithic silicon substrate of the host

logic device such as, a micro-controller, digital signal processor, and application-specific integrated circuit [29]. It has undergone tremendous growth of demand as compared to the stand-alone flash memory (self-contained and uses a separate chip area for flash-memory-related operations) in the past 20 years [30]. An embedded flash memory help to achieve a significant reduction in the power consumption as compared to the stand-alone flash memory. This is because, it eliminates the additional circuitry, buffers and the bus connection required for the communication between the stand-alone flash memory and the host logic device. A system on a chip can be realized with the embedded flash memory, which is not possible with the stand-alone flash memory [29].

An embedded flash memory consists of an array of memory cells that store the information. These memory cells are made from the floating gate transistors. Significant research efforts have been made in the past few years in designing these flash cells suitable for embedded applications. Some popular examples are, the source-coupled split-gate (SCSG) cell [31] for low cost manufacturability, the 2TS cell [32] for low-voltage operation, and the MoneT cell [33] for high-density and high-speed operation. The 2TS cell works on low operating power in both, active and standby modes, and allows a flash module to be designed into a single-cell battery powered system. It also offers enhanced read access at low VDD while maintaining low power consumption [32]. Therefore, an integration of the 2TS cell with the designed data-flow architecture will result in a power-efficient processor for the event-driven applications. Also, the die size of the designed architecture will not significantly increase due to the embedded flash, as the architecture with its basic set of function does not require large amount of flash memory.

The embedded flash memory has a widespread application in several devices such as micro-controllers, application-specific integrated circuits, and digital signal processors. The embedded flash memory is used to store parameters, coefficients, and lookup tables in micro-

controllers and application specific integrated circuits. It is also useful for code storage purpose, that includes a set of configuration registers to configure the host logic device for implementing specific types of operations [29]. The embedded flash memory cell arrays are used to store reset configuration information, which are later retrieved by the host logic device at power-up or reset [29]. A 65-nm 2T-embedded flash memory implementation for a high reliability System on Chip application proposed in [34] is a good illustration of the high performance of embedded flash memory on a small feature size process technology.

This thesis proposes the integration of low power (2TS), embedded flash memory cells in the data-flow architecture to achieve significant power savings. The non-volatile nature of these flash memory cells allows the configuration registers in the architecture to retain the information required to perform specific predefined functions, during the powered-down state. This facilitates the power-down of sections of the architecture when not in use (idle state) and results in significant reduction in leakage power dissipation.

Chapter 3

MDE-based Application Framework

This chapter describes the design of an MDE-based application framework that is integrated with the automation framework developed in [6]. The application framework makes use of MDE principles to achieve simple application programming and automated validation of the implementations. Section 3.1 gives an overview of the entire automation framework. Section 3.2 describes the design of the application modeling framework in detail. Section 3.3 describes the modifications done to the validation framework to achieve easier application validation.

3.1 Overview of the Automation Framework

This section gives an outline of the automation framework highlighting the contributions of this thesis. The existing automation framework is an integration of the preliminary work described in [6, 8], scheduler implementation described in [35], and the contributions of this thesis. The top-level flow of the components of the framework is illustrated in Figure 3.1.

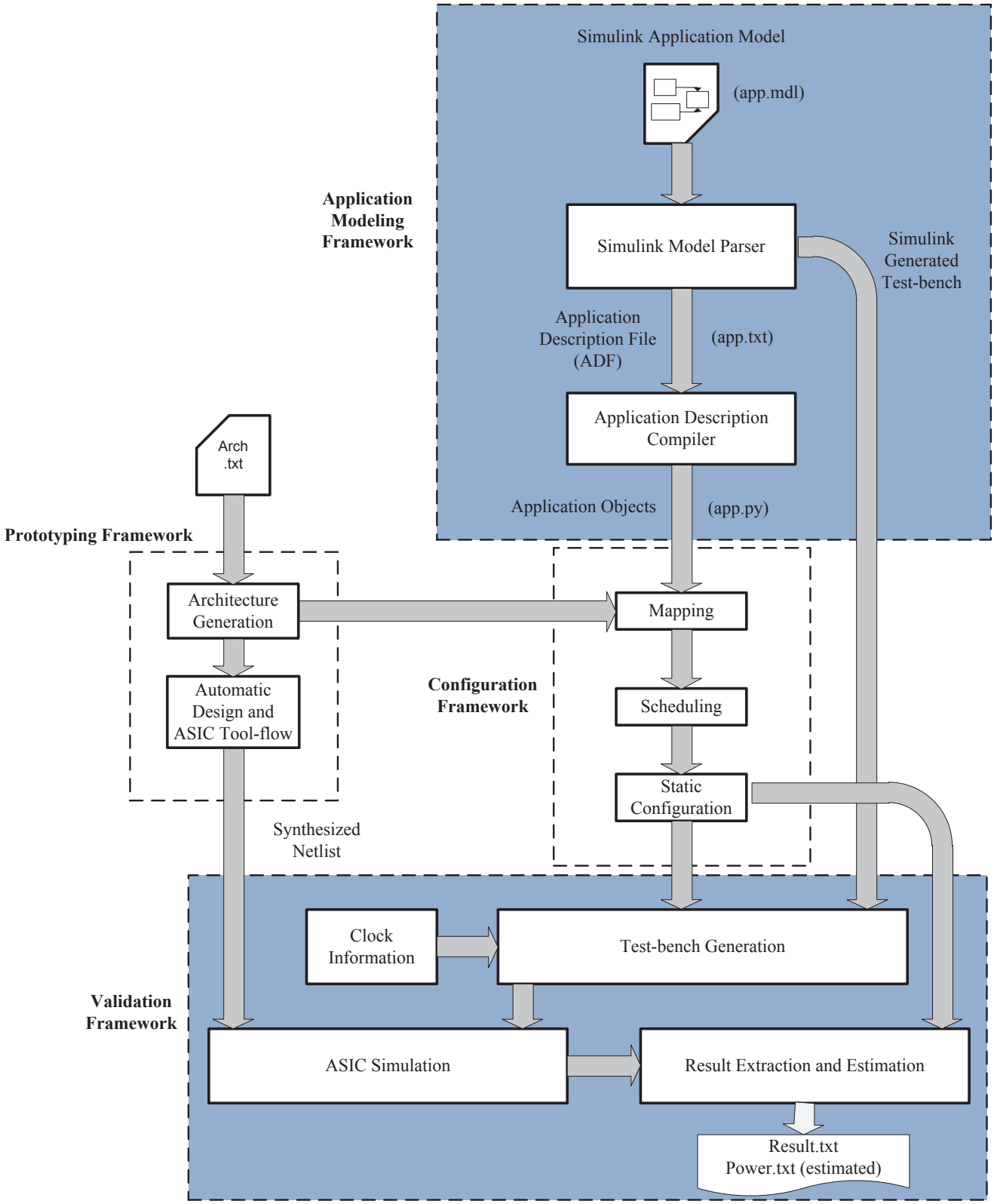

Figure 3.1: Automation Framework Outline

As shown in Figure 3.1, the automation framework consists of four main components, an application framework, a prototyping framework, a configuration framework, and a validation framework. The primary inputs to the automation framework are Simulink application model and the base hardware architecture description file. The application framework simplifies the application development by using Simulink, an MDE-based commercial software package [9]. The input applications are created in Simulink design environment using predefined Simulink blocks, subsystems, and user-defined embedded MATLAB functions. The prototyping framework allows rapid prototyping of an architecture instance from the input high-level architecture description files [6]. Additionally, the hardware design files for the description are generated, synthesized, placed, and routed through an automated Synopsys ASIC tool-flow chain. The configuration framework further maps the application graph onto the matching coarse-grained modules in the architecture and generates a static schedule of the application-architecture instance. [6]. The validation framework accelerates the validation of the desired application on the target architecture. It uses the application model simulation in Simulink to generate test-input packets to validate the application execution on the architecture.

The details of the configuration and the prototyping framework are provided in [6]. The implementations of the application framework and the validation framework are explained in detail in the following sections.

3.2 Application Framework Design

This section describes the implementation details of the application framework. The initial version [6] of the automation framework required the input applications to be written as high-level Python objects. The internal configuration of the application nodes were provided

through the object arguments. Additionally, python scripts were written to generate the test-stimulus to validate the application execution on the processor architecture. The application framework designed in this thesis simplifies the application modeling procedure. It aids in programming applications at a higher level of abstraction, resulting in easy configurability, minimized error, and meaningful validation. As shown in Figure 3.1, the framework consists of three main components, an application representation in Simulink, a Simulink model parser, and an application description compiler. These components are explained in detail in the following sections.

3.2.1 Application Representation in Simulink

The applications are developed in the graphical environment of Mathworks Simulink. Simulink offers a ready-to-use library with a variety of predefined blocks such as adder, subtracter, filter, and analog to digital converter. Additionally, Simulink offers a subsystem [36] in which a set of predefined blocks are replaced by a single block. It can be used to represent certain functional units of the architecture, such as analog to digital converter and timer, by grouping predefined blocks into a subsystem. It also provides the MATLAB Function block [37] to simulate a user-defined block functionality that is not provided by the predefined library and subsystem.

Figure 3.2 shows the representation of a *3*-coefficient filter application in Simulink. As shown in the figure, the Multiplier and Adder units are predefined blocks from the Simulink library. The Analog to Digital Converter and Timer blocks are constructed as subsystems, whereas the Splitter, Delay-generator, and Constant units are defined as user-defined embedded MATLAB Function blocks. Some of the blocks, such as Constant and Analog to Digital Converter, are required to be configured with specific parameters for their desired

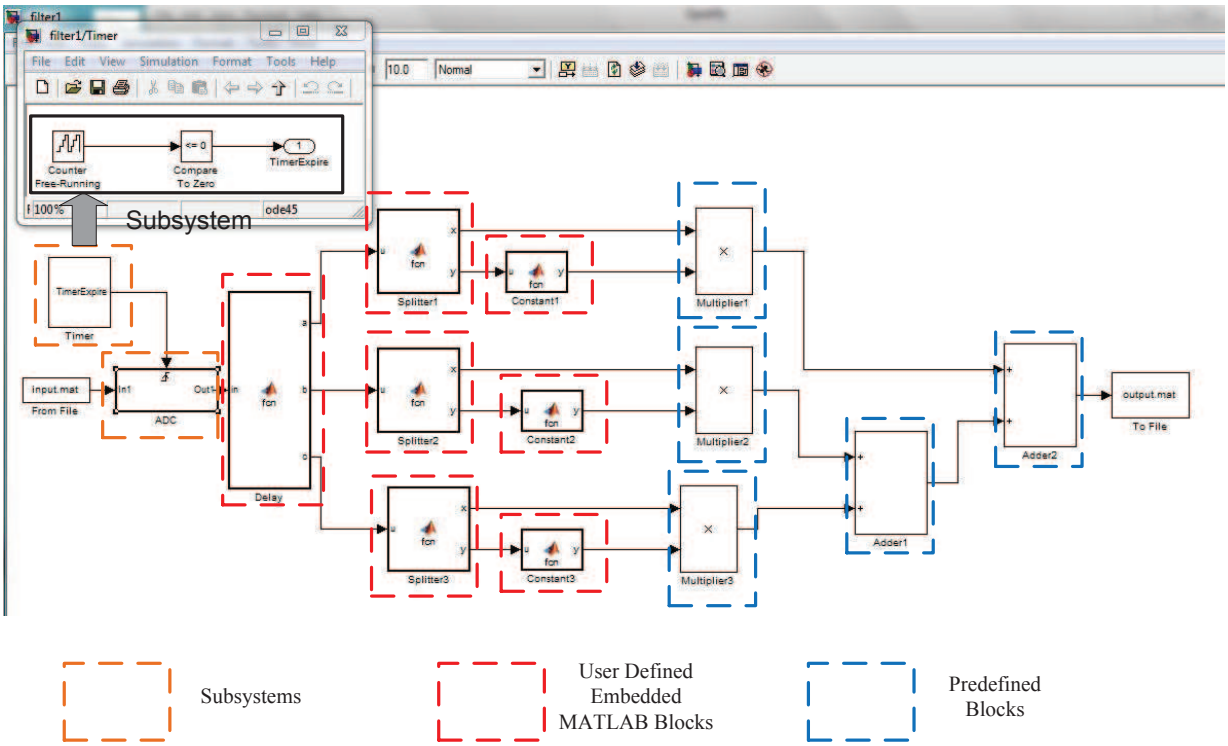

Figure 3.2: Simulink Representation of an Application (3-coefficient filter)

execution. These parameters for the units are set as block configurations in Simulink. The test-stimulus is generated from *input.mat* and *output.mat* after the Simulink model simulation. These files contain a sequence of samples, with each sample consisting of a time stamp and an associated data value.

A set of rules are required to be followed for modeling an application in Simulink. The structure of the application block diagram should follow the logical data-flow path of its execution on the architecture to allow automatic conversion into application graph. The blocks of the model can only by constructed using predefined blocks, subsystems, or embedded user-defined blocks. If an embedded user-defined block is being used, its configuration needs to be supplied externally. A strict naming convention needs to be followed for defining

the Simulink blocks. All the block names should have the exact name string as that of the functional units of the architecture.

3.2.2 Application Model Parser

When a Simulink block diagram is created, its model (.mdl) file is generated automatically in the background. A model file is a structured ASCII file that contains *two* elements, keywords and parameter-value pairs that describe the model. The file describes the model components in a hierarchical structure as shown in Figure 3.3.

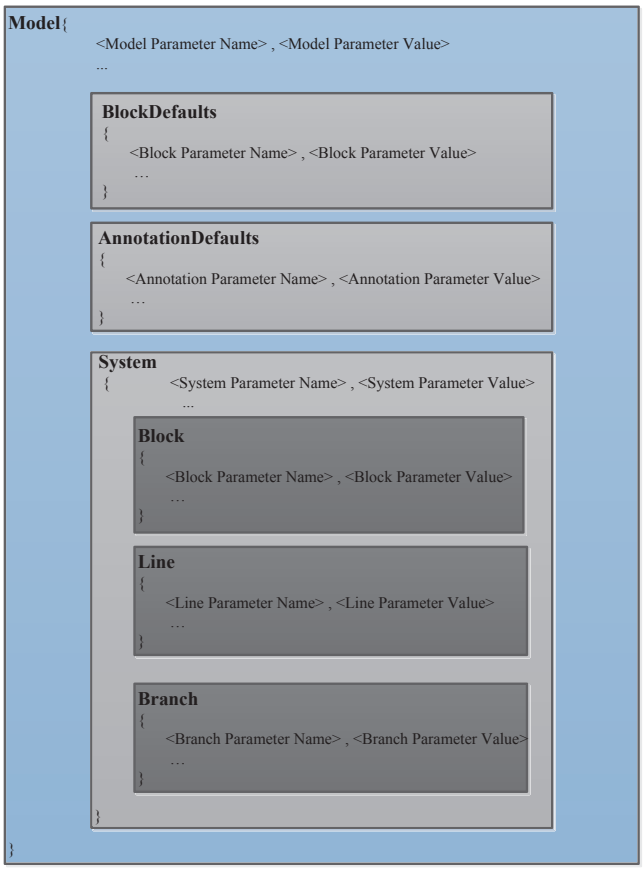

Figure 3.3: Structure of Simulink Model file

The *Model* section, located at the beginning of the model file, defines the values for model-level parameters. These parameters primarily include the model name and simulation parameters. The model name is extracted from this section as the name of the application. The *BlockDefaults* section appears after the simulation parameters and defines the default values for the block parameters within this model. These values can be overridden by individual block parameters, defined in the *BlockSections*. The Simulink block names and their corresponding parameters are extracted from this section of the model file to gather the application node and the internal configuration information. The top-level system and each subsystem in the model are described in a separate *System* section. Each *System* section defines system-level parameters and includes *Block*, *Line*, and *Annotation* sections for each block, line, and annotation in the system. Information of the interconnections between the application nodes is extracted from each *Line* section.

This model file is supplied as an input to the application parser developed in JAVA. The parser makes use of the Simulink JAVA library developed in [38]. The necessary application model information such as the name, block names, parameters, and block connections is extracted from the sections of the model file. The application description file (ADF) containing the application information is built and forwarded to the application description compiler.

3.2.3 Application Description Compiler

A sample application description file (ADF) generated by the Simulink model parser is shown below. It consists of five sections, *application-name*, *application-nodes*, *module*, *internal-configuration*, and *connections* as shown. The *application-nodes* section lists the number of times a module has been reused by an application. The *module* section specifies the modules of the architecture used by an application. The *internal-configuration* section lists the con-

figuration values specific to the modules of an architecture. The *connections* section specifies the interconnections between the application nodes.

```
application_name = ["3-coefficient filter"]

application_nodes = ["Multplier1", "Constant1", "Multiplier2", "Adder1", "Splitter2",
                     "Multiplier3", "Delay", "ADC", "To File", "Constant3", "Constant2",
                     "Timer", "Adder2", "Splitter3", "Splitter1"]

modules = ["Multiplier", "Constant", "Delay", "Adder", "Timer", "ToFile", "ADC", "Splitter"]

internal_configuration = [{"Constant1":0.2}, {"Constant2":0.1}, {"Constant3":0.5},
                          {"Timer":30,000}, {"ADC":8}]

connections = [[Timer, ADC], [ADC, Delay], [Delay, Splitter1], [Delay, Splitter2],
               [Delay, Splitter3], [Splitter1, Constant1], [Splitter1, Multiplier1],
               [Splitter2, Constant2], [Splitter2, Multiplier2], [Splitter3, Constant3],
               [Splitter3, Multiplier3], [Constant1, Multiplier1], [Constant2, Multiplier2],
               [Constant3, Multiplier3], [Multiplier1, Adder1], [Multiplier2, Adder1],
               [Multiplier3, Adder2], [Adder1, Adder2], [Adder2, To File]]
```

The ADF is given as an input to the application description compiler developed in Python. The complier transforms the ADF producing python objects for the application as a directed acyclic graph (DAG). It is given by App_G = (V_{app}, E_{app}), where V_{app} is the set of application nodes and E_{app} is the set of interconnections (data-flow) between the nodes [6]. The generated application graph is forwarded to the configuration framework for deriving the static execution schedule on the processor architecture [6]. An example application graph for a *3*-coefficient filter is shown in Figure 3.4.

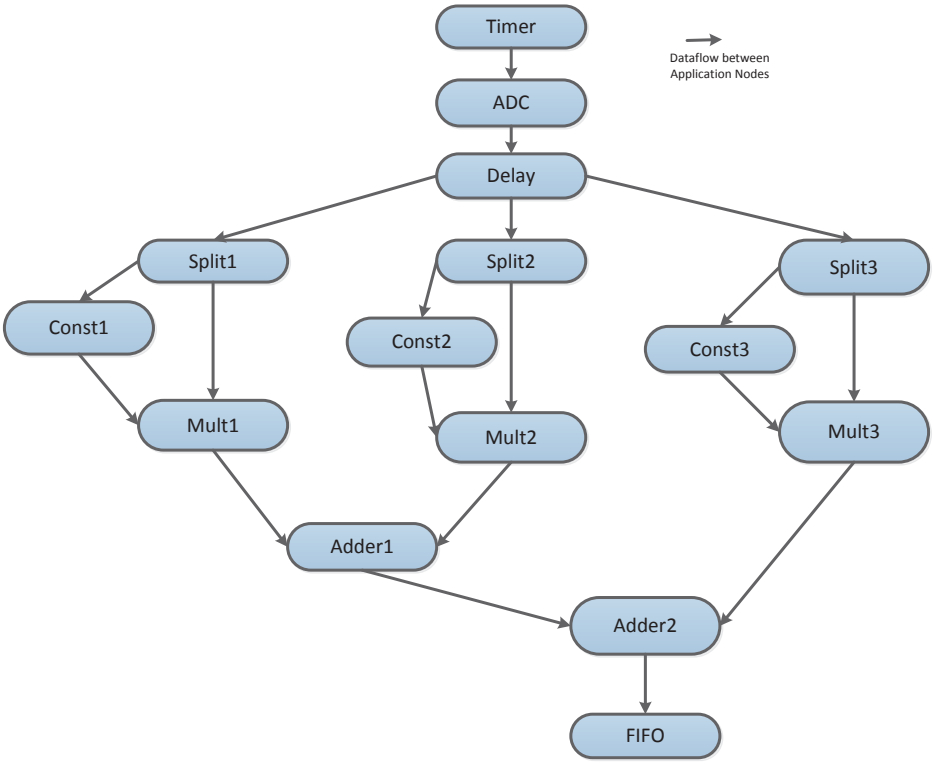

Figure 3.4: An Example Application Graph (Application-III (3-coefficient filter))

3.3 Validation Framework Design

The final component in the automation framework is the validation framework. The initial version of this framework is implemented in [6]. The work done in this thesis simplifies the validation procedure using the test-stimulus generated from the Simulink block diagram. As shown in the Figure 4.1, the validation framework takes in *4* inputs, the hardware-design, the static configuration, Simulink generated test-stimulus, and the clock information as input. The validation framework has three steps, automatic test-bench generation, ASIC simulation and results extraction and estimation.

3.3.1 Automatic Test-Bench Generation

This section describes the automation of the test-bench generation using the Simulink test-stimulus. The static application schedule produced by the configuration framework is verified by automatically generating a test-bench; using the Simulink generated stimulus and the clock information. The primary goal of the test-bench is to supply the architecture with the generated configuration values as well as external port inputs and the functional units specific configurations. In the first step, the configuration file generated by the scheduler [35] is read and is used to produce the configuration packets in the architecture defined packet format [6]. Apart from these execution specific configurations, the test-bench also needs information about the functional unit specific configurations and the external port inputs. This information is obtained from the Simulink generated test-stimulus (input.mat and output.mat). It also needs the clock information which is provided externally. The framework uses this information to generate a Verilog test-bench module that supplies the wrapper configurations, internal configurations, and test data to the architecture design for simulation [6].

3.3.2 Simulation, Results Extraction and Estimation

This section gives a brief overview of the simulation and result extraction framework built in [6][8] and explains the contributions of thesis in implementing the power estimation framework for the flash-based architecture. In the first step, the generated test-bench and the design net-list are used to simulate the application on the architecture. The application is simulated using the Synopsys VCS tool [39]. The area and timing results are automatically extracted from the simulation output.

Additionally, the power consumption for executing an application on a given architecture

instance is obtained from the Synopsys PowerTime-PX tool [39]. A gate-level power analysis is performed by the tool, and power consumption of every hierarchical module is reported. A detailed power report that has information of dynamic and static power consumption for all the levels of hierarchy is obtained from the PowerTime-PX analysis. The initial version of the framework utilized 90-nm technology from Synopsys that has capabilities to support the leakage aware design. This thesis replaces the 90-nm library with a 45-nm TSMC technology library [40] that does not have a power-aware capability to power-down the functional units of the architecture during the idle cycles. However, the static schedule of the application execution allows the framework to determine the cycles for which a functional unit needs to power-down. This schedule is mapped onto the generated power report and the power consumption in each cycle is estimated by considering *zero* power dissipation in the functional units that are considered to be switched off in that cycle. The framework provides an estimate of the power consumption as an output along with the actual area results.

The flash memory functionality in the architecture is described in Chapter 4. The assumptions taken into consideration while estimating the power savings are explained in Chapter 5.

Chapter 4

Flexible Design of the Processor Architecture

This chapter describes the design of the data-flow processor architecture, giving an overview of the previous implementation and highlighting the contributions of this thesis. A brief outline of the architecture functionality is provided in Section 4.1. Section 4.2 describes the flexible and scalable communication packet structure and parameterized functional unit template. Section 4.3 discusses the use of low power flash memories for leakage power reduction.

4.1 Overview of the Processor Architecture

A data-flow processor architecture without an instruction set was implemented in [8], specifically targeted for event driven applications. It offered asynchronous event handling and resource-sharing using data-flow principles. The data-flow architecture exploration space is

expanded further in [35] with an implementation of a multi-bus architecture. Figure 4.1 gives a high-level representation of an instance of the multi-bus processor architecture.

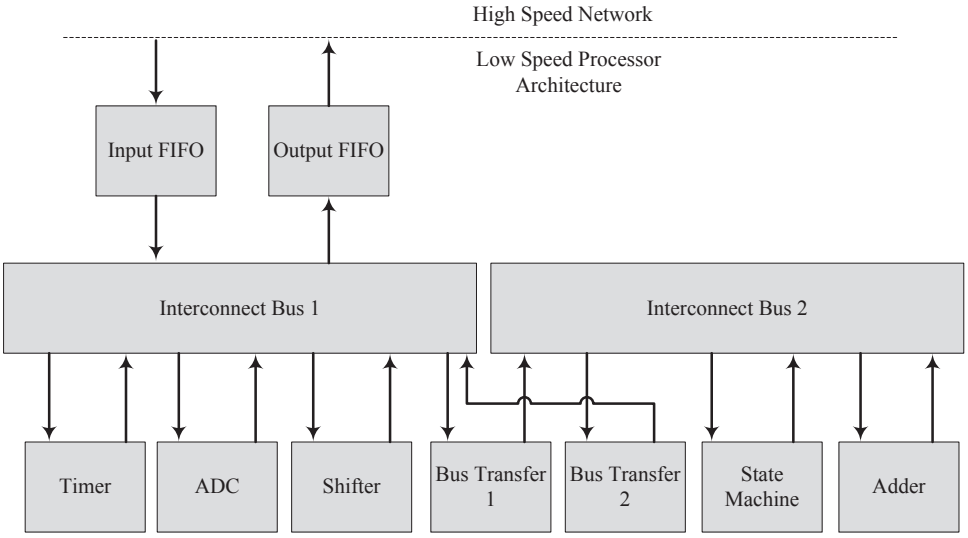

Figure 4.1: A High-Level Representation of a Multi-bus Architecture

As shown in Figure 4.1, the main components of the processor architecture are FIFOs, functional modules and event buses. A FIFO is used for communicating with the network to synchronize the flow of events between the low speed processor and the high speed network. The FIFOs act as configuration channels, transferring the configuration events from the outside network to the functional units. The functional units are coarse-grained configurable modules that perform a predefined function [8]. The functionality of the coarse-grained unit is calibrated to the application specific behavior through configuration events [6]. The functional unit starts executing only after it receives all the inputs and generates output data-events. The functional units are interconnected through two event buses network (multiple buses are possible) that transfer the data-events between the functional modules [35]. The output data-events are transferred to their destination on predefined cycles through the event-bus network.

There is no central control unit for sequencing the execution of functional units. Each functional unit executes autonomously if it receives the required input. The flow of data (also called events) between the functional units facilitates the execution of an application on the architecture. This ensures that the architecture does not suffer from run-time resource conflicts, giving predictable performance. The details of the multi-bus architecture implementation and functionality are provided in [35].

4.2 Scalable and Flexible Architecture Design

This section describes the implementation of a scalable and flexible design of the architecture. The implementation builds upon the initial version of the architecture [8] making it suitable for a wider range of applications. Section 3.2.1 describes the design of the communication packet structure for transmission of configuration and data events. Section 3.2.2 illustrates the template design of the functional units of the architecture.

4.2.1 Communication Packet Structure

This section illustrates the flexible packet structure design for the architecture. The communication packet structure is an important factor in the design of the architecture, as the capabilities and limitations of the architecture can be drawn from it. The initial version of the architecture was designed to handle a fixed packet length of twelve bits for both configuration and data. The fixed packet structure restricted the number of modules in the architecture to eight. It also introduced limitations on the configuration data that can be transferred in a single packet. This limited the class of applications that can be successfully executed on the initial version.

The existing version of the processor architecture is designed to handle a flexible packet structure facilitating scalability and adaptability in terms of the number of modules and configuration data it can hold. The redesigned packet structure is illustrated in Figure 4.2.

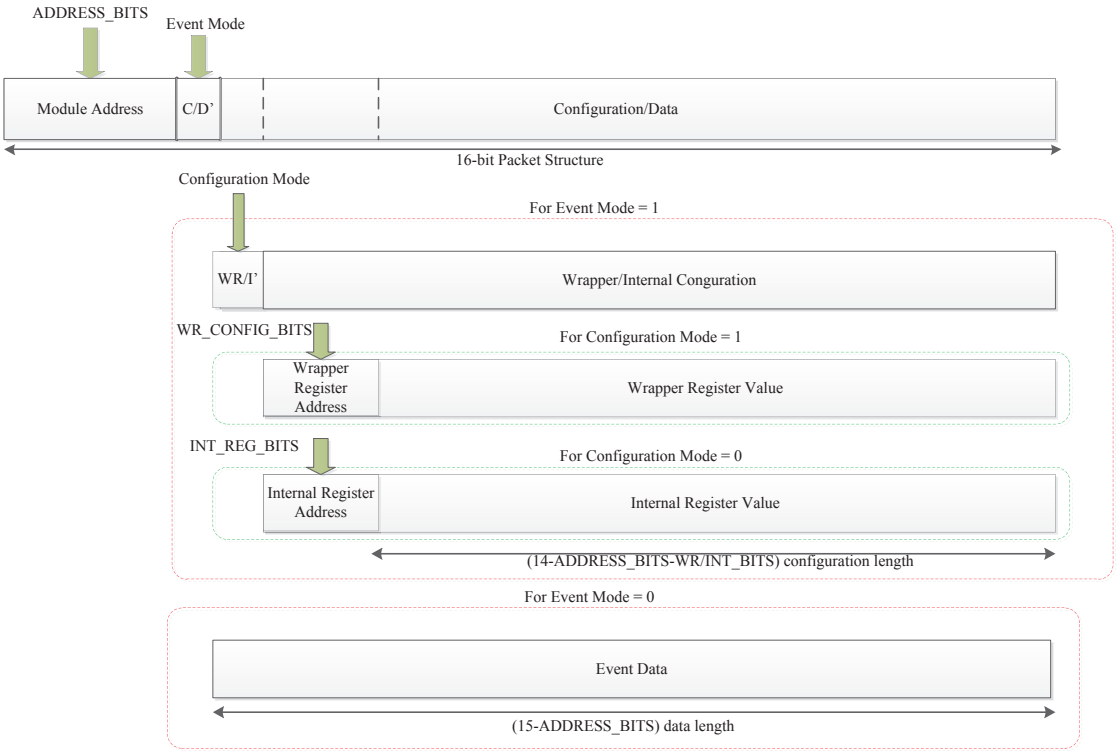

Figure 4.2: Communication Packet Structure for the Architecture

As shown in Figure 4.2, the communication packet length is set to *16*. There are three parameters, ADDRESS_BITS, WR_CONFIG_BITS, and INT_REG_BITS that determine the packet structure. These parameters are customizable according to the application requirement through the software framework. The framework provides these parameters as arguments to hardware design files before compilation. The ADDRESS_BITS parameter controls the number of address bits allotted to the hardware functional units of the architecture. The bit next to the functional unit address is termed as *Event Mode* bit. If this bit is high, then the packet contains a configuration event for a specific module within the architecture. If

this bit is low, then the packet contains a data event generated due to the unit execution. For a packet containing a configuration event, the remaining portion of the packet consists of configuration/internal register addresses, which are controlled by the WR_CONIG_BITS and INT_REG_BITS parameters. These parameters control the number of configuration/internal registers within the architecture functional units. The remaining bits in the communication packet contain configuration values. If the *Event Mode* bit is low, then the entire packet structure after the event mode bit contains the data events flowing among the hardware units of the architecture.

4.2.2 Design of the Functional Units

This section describes the functional unit template design with a focus on the contributions of this thesis. The previous version of the template was developed in [6][8]. This thesis introduces new parameters and modifies the functionality of the template to elevate its flexibility and scalability.

4.2.3 Structure of Redesigned Parameterized Template

The template structure built around the functional units allows them to interact with the interconnection network. The design of the internal module is made independent of the changes in the interconnection network [8]. Such an approach reduces the design effort of a functional unit because an internal module only needs to be plugged into the template. The template is redesigned in a parameterizable fashion facilitating enhanced flexibility. Figure 4.3 shows the parameters and components of the functional unit template.

Input Bus-interface, Output Bus-interface, Wrapper Configuration, and Internal Configuration form the wrapper of a functional unit. The internal module executes the module specific

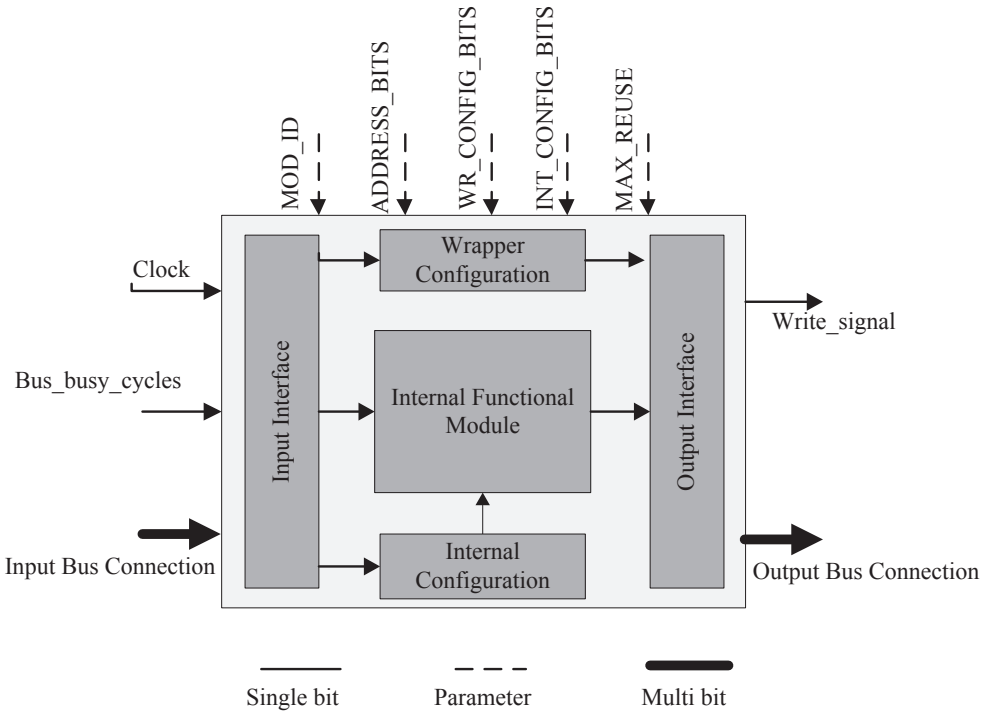

Figure 4.3: Parameterized Template for the Functional Units

functionality. The modules of a functional unit take multiple cycles for execution. Their execution is independent of the interconnection network and the functionality of the internal module. To achieve this independence, handshake signals are employed for signaling the completion of execution of a module, as shown in Figure 4.3. The template is parameterizable, with control over the functional unit address, configuration/internal registers, and the data length. A brief explanation of the new parameters is as follows. The description of the remaining parameters is provided in [8].

- ADDRESS_BITS: This parameter controls the number of functional units in an architecture instance. This parameter is useful particularly when an application requires a significant (more than *8*) number of functional units for its execution. The default value of this parameter is set to 4.

- WR_CONFIG_BITS: This parameter controls the number of wrapper configuration registers in a functional unit. This parameter is particularly useful for complex applications that demand a significant number of output registers. The default value of this parameter is set to 3.

- INT_CONFIG_BITS: This optional parameter controls the number of functional unit specific internal configuration registers. This parameter is optional because every functional unit does not require configuration registers.

4.2.4 Input Bus-Interface Module

This section describes the new functionality introduced in the input bus-interface module to read a flexible packet structure. Its block diagram is shown in Figure 4.4.

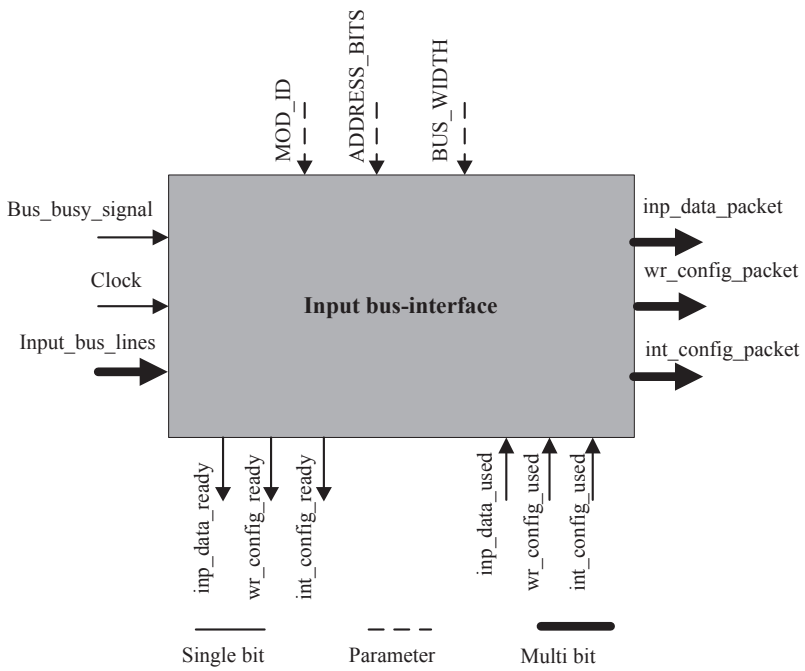

Figure 4.4: Input Bus-interface Structure for a Functional Unit

The input bus-interface polls the *input_bus_lines* for an input event. Once an input is received, the BUS_WIDTH parameter determines number of cycles required to form an event packet. The destination address of the event is transferred onto the bus before the data. The number of bits read as the destination address is dependent on the ADDRESS_BITS parameter. After the destination address is read, it is compared with the MOD_ID, and the input packet is accepted by the only functional unit with the same MOD_ID. The input bus-interface reads the next bit (Event Mode) and determines whether the received packet is either configuration or data. It forms a configuration or data sub-packet and forwards it to the wrapper or the internal module respectively.

4.2.5 Wrapper Configuration Module

This section describes the new functionality introduced in the wrapper configuration module to increase the number of configuration registers. The overall structure of the wrapper configuration block is shown in Figure 4.4.

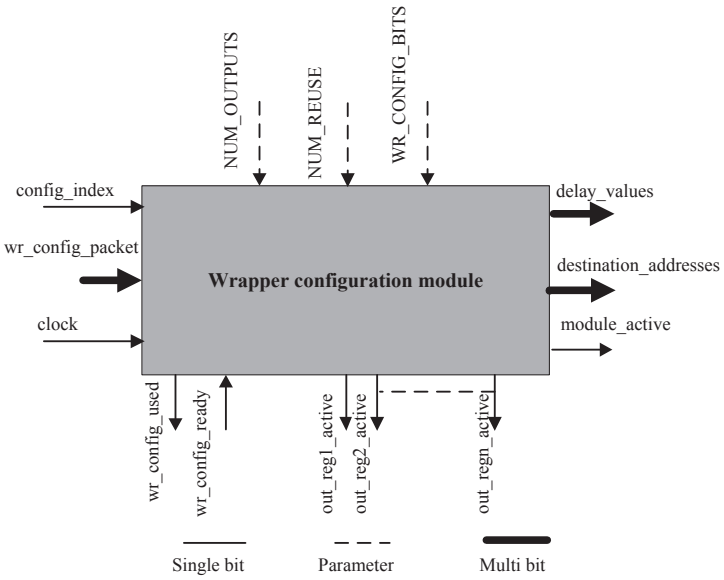

Figure 4.5: Wrapper Configuration Module of a Functional Unit

The wrapper configuration registers are common to all of the functional units. These registers primarily contain destination and delay information derived from the application schedule. The number of configuration registers in the module is dependent on the WR_CONFIG_BITS parameter. This parameter is set according to the number of output registers reused. The details of multiplier output registers and the handshaking signals are provided in [35]. The configuration register at the first address indicates whether a specific functional unit is ready for operation. The configuration register at the second address signifies the number of times a functional unit is reused (less than MAX_REUSE). The remaining configuration registers specify the destination and delay values for each output register.

4.2.6 Internal Configuration Module

This section describes the new functionality introduced in the internal configuration module to allow a variable number of internal configuration registers. The overall structure of the internal configuration block is shown in Figure 4.4.

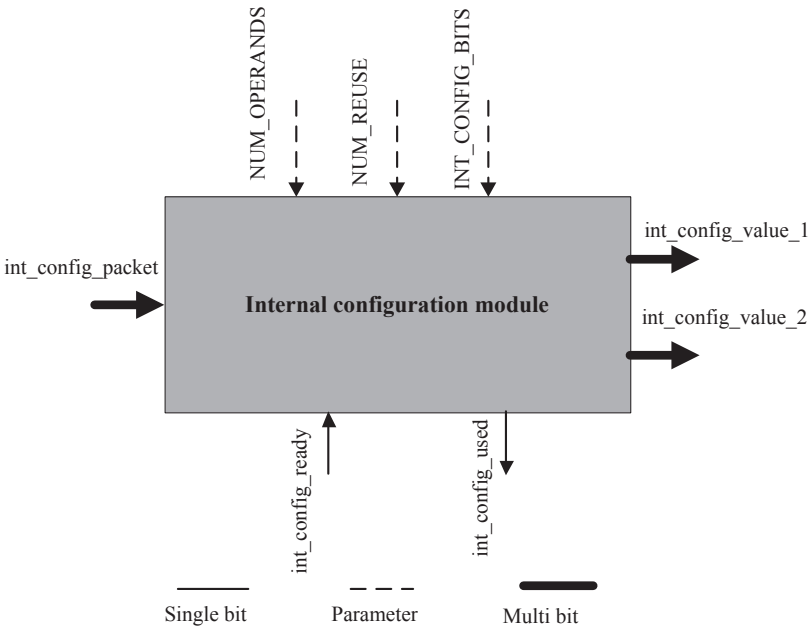

Figure 4.6: Internal Configuration Module of a Functional Unit

The internal configuration module receives the configuration packet from the input bus interface. The INT_CONFIG_BITS parameter determines the number of internal configuration registers, which are different for different functional units. The remaining bits in the packet structure contain configuration values and are allotted to specific registers. These configuration values are forwarded to the internal module of the functional unit as shown in Figure 4.3.

4.3 Power-aware Design

This section describes the power-aware design of the processor architecture that targets static power consumption. In the initial version of the architecture, a power-aware design was achieved by switching off the power to the modules of a functional unit during an application execution [8]. However, the entire functional unit was not powered-down to preserve the

destination and delay values in the wrapper configuration module of the functional units. This lead to significant leakage power consumption during the idle cycles [8].

This thesis proposes the use of embedded flash memory cells as a replacement for the CMOS static-RAM cells in the wrapper configuration module. Flash memory stores information in an array of memory cells made from floating-gate transistors [41]. These cells can be used to store configurations, allowing the functional unit to be powered-down during idle execution cycles. The 45-nm standard cell library used for the ASIC implementation does not have support for the embedded flash memory cells. However, an estimation of the power savings can be made by assuming that the CMOS static-RAM register cells are replaced by the flash memory cells in the wrapper configuration modules of the functional units. The effectiveness of flash memory in the processor architecture is determined using the result extraction and estimation framework. The power analysis focuses on the comparison of the power savings achieved in the flash-based architecture with the power consumption in a non-flash based architecture.

As shown in Table 4.1, in the flash-based architecture, there are *three* primary phases in the application schedule, the configuration phase, the execution phase (busy cycles), and the idle phase (free cycles). During the configuration phase, the input bus-interface and the wrapper configuration module are powered-up to transfer the configuration events, and the internal and the output bus-interface modules are powered-down. After the configuration phase is over, the wrapper configuration module is powered-down.

During the application execution phase, the internal and the output bus-interface modules are powered-up when the input bus-interface receives all of the inputs required for the functional unit execution. The wrapper configuration module is powered-up to transfer configuration events to the output bus-interface and is powered-down after the transfer is complete. The internal module is powered-down when the results are transferred to the

Table 4.1: Execution Sequence of Flash-based Architecture

Power States in Different Phases		Description
Phase	(input, wrapper, internal, output)	
Configuration	(ON, ON, OFF, OFF)	the wrapper configuration module reads from the input bus-interface
Execution	(ON, OFF, OFF, OFF)	the input bus-interface reads from the bus
	(ON, OFF, ON, OFF)	internal module reads from the input bus-interface
	(OFF, OFF, ON, OFF)	the internal module execution
	(ON, ON, ON, ON)	simultaneous execution of all the modules
	(OFF, OFF, ON, ON)	the output bus-interface reads from the internal module
	(OFF, OFF, OFF, ON)	output bus-interface transfers the result on the event-bus
Idle	(OFF, OFF, OFF, OFF)	the entire unit is powered-down

output bus-interface, and no more inputs are available. Similarly, the output bus-interface is powered-down when a data packet is transferred onto the bus, and there are no more results from the internal module. Because the wrapper configuration module is powered-down when not in use, significant power savings are achieved in the execution phase of the flash-based design as compared to non-flash based architecture.

In the idle phase, the input bus-interface in powered-down. The input-FIFO signals the input bus-interface of the functional unit to power up for the next execution based on the configurations (free cycles) stored in it [8]. The wrapper configuration module, the internal unit, and the output bus-interface stay powered-down and therefore, the entire functional unit is switched off during this phase. This results in significant power savings in the idle phase as well.

Similarly, flash memory cells can be used to replace the configuration registers in the input-

FIFO and power-down the components of the FIFO when not in use. The sections of the input FIFO are powered-down after the configuration phase, and are powered-up when there are events from the network to the processor architecture during execution. The output FIFO is always in a powered-down state, and it is powered-up when there are events from the processor architecture to the network. The different power states of input and output FIFO are provided in Appendix A.3. The detailed power analysis of the flash-based architecture is provided in the next chapter.

Chapter 5

Experimental Results

In this chapter, the contributions made to the architecture design and the associated software framework are validated by executing an application suite on the representative architecture instances. In Section 5.1, the development of the application suite in Mathworks Simulink environment is validated. In Section 5.2, the scalability and the adaptability of the re-designed architecture are demonstrated. In Section 5.3, the power savings achieved due to the embedded non-volatile flash memory cells are presented.

5.1 Application Modeling Framework Validation

The application modeling framework is validated by using it successfully to transform the Simulink representation of each application of the suite into the corresponding application graph. The transformation is performed in two stages. In the first stage, the Simulink application model is converted to the corresponding application description file (ADF) by the application framework. In the second stage, the ADF is transformed into application graph. The experimental application suite in Simulink used for the processor architecture

evaluation is presented in 5.1.1. The representative instances of the architecture are described in 5.1.2. The experimental setup is explained in section 5.1.3. The Simulink representations of the entire application suite and the generated application graphs are provided in Appendix A and B, respectively.

5.1.1 Experimental Application Suite

The micro-controller application range is broad, with numerous vendors, technologies, and markets included in it. However, this wide range can be categorized on the basis of specific characteristics of the applications and the field of use, such as industrial automation, digital signal processing, robotics, and sensor driven systems. An application suite is developed in Simulink for the evaluation of the dataflow processor architecture, including representations of these popular PIC micro-controller application fields. In sensor driven systems, each type of sensor (e.g., accelerometer, infrared sensor) and associated applications present different requirements to the processors in terms of their computation. Therefore, some of the applications in the suite represent the processing challenges these sensors introduce. The application suite is divided into subsets based on the specific application characteristics. Table 5.1 summarizes the applications in the suite and the reasons for their selection. The execution behaviour of these applications on the architecture enables analysis of the dataflow processor performance as a target platform for the real-world applications.

- Application-I set consists of a temperature controller (Application-I) [42] with a simple logic-control mechanism. The temperature controller allows the user to change the threshold temperature through an asynchronous event from the network. It consists of an ADC which samples a sensor value periodically based on a timer event. The output

of the ADC is compared with a specified temperature threshold. The difference is fed to a state-machine that controls the switch-on and switch-off of an actuator.

- Application-II set consists of variations of a 2-operand multiplier, in terms of the operand size (2-bit, 4-bit, 6-bit, 8-bit). A multiplier application makes use of the shift-and-add approach to multiply the two operands, and thus, demands extensive resource sharing [35, 43].

- Application-III set consists of variations of a discrete-time, finite impulse response (FIR), n-coefficient digital filter application [44]. Its output is a weighted sum of the current value and a finite number of previous values of the input.

- Application-IV set is an application that determines the square root of an integer by the method of successive approximation [45]. Square root calculation is chosen because it plays a significant role in the digital signal processing applications [46].

- Application-V set consists of a free-fall detection application [47] that protects the data in the hard drives from a free-fall. It consists of an ADC which continuously takes analog inputs from a 3-axis accelerometer [48] and calculates a g-value based on standard computations. The calculated g-value is compared with a predefined threshold. If the predefined threshold is crossed, the actuator is switched on.

- Application-VI set is an application that assists a robot to follow a predefined path [35]. This application consists of an ADC which takes inputs from two sensors attached to two limbs, and the sensor variations are fed to a state-machine. The state machine directs the robot to turn left or right through an actuator.

Table 5.1: Summary of the Application Suite

App No.	Application Name	Field/Feature Represented	Application Characteristics			Appendix	
			Module Types	Nodes	Edges	Model	Graph
I	Temperature controller	Control based automation	7	9	11	A.1	B.1
II	Multiplier (*2*-bit)	Mathematical routine functions	8	17	22	A.2	B.2
	Multiplier (*4*-bit)		8	35	46		
	Multiplier (*6*-bit)		8	53	76		
	Multiplier (*8*-bit)		8	71	94		
III	*2*-coefficient FIR Filter	Digital signal processing	8	11	13	A.3	B.3
	3-coefficient FIR Filter		8	15	19		
	8-coefficient FIR Filter		8	35	49		
IV	Square root	Non-linear functions	8	28	39	A.4	B.4
V	Free-fall detector	Interfaced with an accelerometer sensor	9	32	46	A.5	B.5
VI	Path follower	Micro-controller based robotics, Interfaced with infrared sensor	9	23	29	A.6	B.6

5.1.2 Architecture Set

Each application in the suite has a different architectural requirement. Therefore, six architecture sets are used for the experiments, each deferring from the rest with respect to the number/types of the functional units in the architecture. The functional units, including, Timer, ADC, Splitter, State Machine, Adder, Subtractor, FIFO, Constant-generator,

Table 5.2: Summary of the Architecture Set

Set No.	Feature/Tradeoff Demonstrated	New Functional Units Introduced	Application Suite Support	Appendix
I	Area and energy overhead	-	Application-I	C.1
II	Support for complex (number of nodes) applications	-	Application-II variations	C.2
III	Support for multiple output registers	Delay-generator	Application-III variations	C.3
IV	Support for larger wrapper configuration	Multiplier	Application-IV	C.4
V	Support for 8+ functional units	Comparator	Application-V	C.5
VI	Support for larger internal configuration	-	Application-VI	C.6

and ALU are presented in [8]. Certain functional units are designed as a part of this work and are mentioned in Table 5.2. Table 5.2 summarizes the architecture set specifying the feature/tradeoffs demonstrated by each instance and the support for the application suite.

5.1.3 Experimental Setup

The Simulink application model and the architecture description file are given as inputs to the programming framework. The framework automatically maps the application onto the architecture and generates a static schedule of the execution for the hardware functional units. The architecture designs are provided as an input to the tool-flow automation framework, which performs synthesis, automatic placement, and routing of the architecture instance. A standard cell based ASIC approach is used based upon a 45-nm technology library from TSMC [40]. The verification framework configures the applications onto the architectures and validates the application execution with the help of the Simulink generated

test-bench.

The area of an architecture instance is extracted from the Synopsys IC Compiler [39] which has details of the standard cell and the interconnection areas. The power consumption for executing an application on a given architecture instance is obtained from the Synopsys PowerTime-PX [39]. The inputs for power analysis are the gate-level netlist of the design from the Synopsys IC Compiler, the timing and parasitic information files extracted after the placement and route process, and the waveform output of post-layout simulation. A gate-level power analysis is performed, and power consumption of every hierarchical module is reported. A detailed power report that has information of dynamic and static power consumption for all the levels of hierarchy is obtained from the PowerTime-PX analysis.

The 45-nm technology library used in this approach does not have a power-aware capability (unlike the 90-nm Synopsys technology library [49]) to power-down the functional units of the architecture during the idle cycles. However, the static schedule of the application execution allows the framework to determine the cycles for which a functional unit needs to power-down. This schedule is mapped on the generated power report and the power consumption in each cycle is estimated by considering zero power dissipation in the functional units which are to be switched off in that cycle. The generated results are grouped according to the power states of the functional units. A functional unit has a power manager circuitry to switch off certain modules in a power-aware design. The additional power consumption due to this circuitry is not taken into consideration for this analysis. This is because the analysis concentrates on estimating the power savings achieved by the flash-based approach over non-flash based architecture, and the additional power penalty due to the power circuitry is the same in both the scenarios. This additional power penalty is 10.1 μW per cycle, which is

not significant in complex applications that reuse the functional units in the architecture [8]. The power analysis and the assumptions it is based are discussed in detail in Section 5.3.

5.2 Flexibility of the Architecture Design Template

The packet structure of the architecture is made flexible to provide a control over important architectural design decisions. The advantages of this flexibility and the resulting area and energy penalty are presented in this section.

5.2.1 Scalable Packet Structure

In this section, scalability, in terms of the architectural choices available due to the flexible template design is described. A careful configuration of these architectural choices allows an enhanced support for a wider range of applications. The application suite is executed on the original and the new architecture sets for this section.

The initial architecture implementation had a fixed *12*-bit packet structure with a predefined number of address bits for internal and wrapper configurations, which limited the range of applications that could run be on it. The new architecture implementation consists of a *16*-bit packet structure with a variable number of address bits for functional units, wrapper, and internal configurations. Table 5.3 shows the comparison between the architectural choices in the initial and the new implementations of the architecture.

Table 5.3: Scalability Due to Flexible Packet Structure

Architectural Choice	12-bit Architecture (*fixed*)	16-bit Architecture (*flexible and feasible*)
Maximum no. of functional units	8	8 to 32
Wrapper configuration registers	4	4 to 16
Internal configuration registers	4	4 to 16
Data Length (*bits*)	8	8 to 12

5.2.2 Architectural Support for Application Suite

A direct comparison of the application execution on both versions of the architecture is shown in Table 5.4. It is evident that, as the complexity of an application increases in terms of the specified choices, the initial architecture implementation fails to provide support for the application execution. However, the flexible configurability of these choices allows the new architecture implementation to be a good target for simple as well as complex applications.

Table 5.4: Application Execution Outcome on Initial and New Architecture

Application Instance		New Architecture	Initial Architecture	Reason
I		Supported	Supported	-
II	*2*-bit	Supported	Supported	-
	4,6,8-bit	Supported	Not Supported	Insufficient wrapper configuration registers
III	*2* coeff	Supported	Supported	-
	3,8 coeff	Supported	Not Supported	Insufficient wrapper configuration registers, smaller data length
IV		Supported	Not Supported	Insufficient internal configuration registers
V		Supported	Not Supported	Exceeds maximum allowed functional units
VI		Supported	Supported	-

5.2.3 Area and Energy Cost Analysis of the Redesigned Template

The area and energy overhead of the redesigned template is shown in this section. As discussed in Chapter 4, the functional units are built by plugging the internal modules into the template. The bus-interfaces and the configuration registers form the template of the architecture design. The wrapper configuration and the wrapper counter blocks form the configuration registers. For this experiment, the new architecture implementation is parameterized to allow *16* functional units, *8* wrapper configuration registers, and *8* internal configuration registers.

Application-I, II (*2*-bit), and III (*2*-coefficient) are executed on the initial and new architecture implementations I, II, and III respectively for this experiment. The comparison of the previous template with the current template in terms of their area is shown in Figure 5.1. Note that all of the components of the new architecture occupy more area as compared to the initial implementation. This is a result of the increase in the size of the registers that store the functional unit specific internal configuration and the schedule specific configuration values. An average increase of 23% is noticed in the template component area.

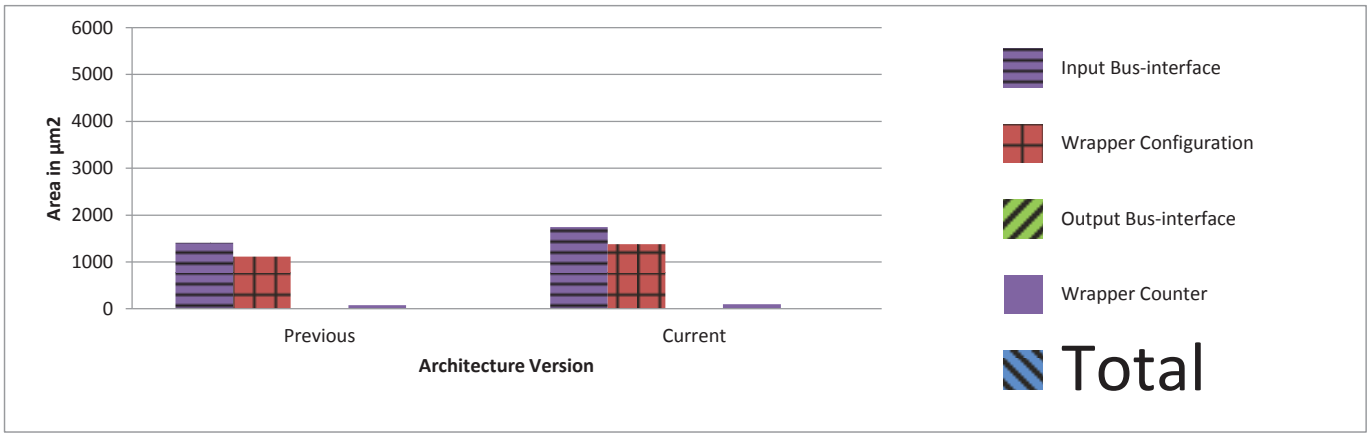

Figure 5.1: Area Comparison of Template Components

The energy consumption comparison of the initial and new template implementation is provided in Figure 5.2. From Figure 5.2, it is evident that the energy consumption of the new template design components is higher (18%) than that of the initial implementation. It is due to the increase in the register sizes of the template components. Thus, there is a trade off between an enhanced support for a broader range of applications and the resultant area and energy overhead.

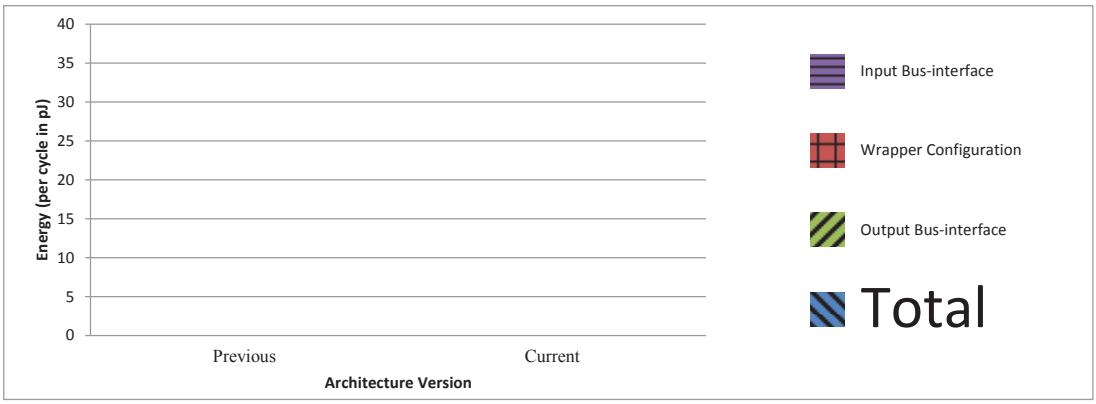

Figure 5.2: Energy Comparison of Template Components

The area of the functional unit as a function of its template and the internal module is presented in Table 5.5. There are three different types of templates. Template-I consists of one input bus-interface and one output bus-interface, Template-II consists of two input bus-interfaces and one output bus interface, and Template-III consists of one input bus-interface and two output bus-interfaces. The comparison of area cost of the templates in both versions of the architecture is shown in Table 5.5. The template area is also compared with the internal module of the respective functional units. It is evident that there is an increase in the internal functioning module area in the current architecture due to the increased size of the registers that store the internal configurations. However, the ratio of the template area to the internal unit area is nearly identical in both the versions.

Table 5.5: Template vs. Internal Module

Function Module	Template Type	Previous Architecture		Current Architecture	
		Template Area (μm^2)	IM Area (μm^2)	Template Area (μm^2)	IM Area (μm^2)
A/D	Template-I	2819.12	1154.23	4063.91	1684
Splitter	Template-II	4228.68	1412.45	5809.45	2109.45
Adder	Template-III	4918.94	1198.34	6382.28	1420.28
Constant	Template-I	2819.12	779.75	4063.91	1140.91
Shifter	Template-II	4251.24	2045.66	5924.45	2628.57
State-machine	Template-III	4918.94	6537.7	6382.28	8767.43

5.2.4 Area of the Template v Packet Structure Width

The parameterized nature of the architecture design enables estimation of the template area for variations in the packet width. This section derives the relationship between the template area and the packet width. From the previous two sections, it is evident that the area of the template varies with the packet width. The width of bus is matched with the packet structure width, simplifying the packet formation logic at the bus-interfaces. This is because the packet capture and the packet transfer blocks require a reduced number of cells for their logic realization [8]. Although the template is designed in a parameterized manner with four primary parameters controlling the packet structure (Section 4.2), the *PACKET_WIDTH* parameter is predominantly responsible for the register sizes in the input and the output bus-interfaces. Therefore, it is possible to estimate the area of the template as a function of the packet width. The estimated area of the template is given by

$$A_{template} = (A_{singleinput}) * w * N_1 + A_{singleoutput} * w * N_2, where, \quad (5.1)$$

$A_{template}$ is the total area of the template,

$A_{singleinput}$ is the average area of a single register input bus-interface in μm^2,

$A_{singleoutput}$ is the average area of a single register output bus-interface in μm^2,

w is the width of the packet structure,

N_1 is the number of input bus-interfaces, and

N_2 is the number of output bus-interfaces.

The values of $A_{singleinput}$ and $A_{singleoutput}$ are calculated from the area results of the new architecture implementation and are shown in Table 5.6.

Table 5.6: Single Register Module Area

Module	Area (μm^2)
Input bus-interface	109.23
Output bus-interface	144.8

Using the equation 5.1, Table 5.7 gives the template area of the architectures with different packet widths. The equation is validated by comparing the calculated template area with the actual area results of the initial architecture implementation. The calculated area is within ≈6% of the actual area.

5.3 Power Estimation of Embedded Flash Memory

This section illustrates the use of embedded flash memory for significant power savings. A prominent leakage power consumption is noticed during the idle execution cycles when the

Table 5.7: Template Area v/s Packet Width

Packet Width	Template Type	Actual Area (μm^2)	Calculated Area
12	Template-I	2819.12	3048.36
	Template-II	4228.68	4359.12
	Template-III	4918.94	4685.96
14	Template-I	-	3598.68
	Template-II		5166.42
	Template-III		5628.02
16	Template-I	4064	4063.91
	Template-II	5811.16	5809.45
	Template-III	6381.28	6382.28
18	Template-I	-	4626.24
	Template-II		6642.0
	Template-III		7236.16

functional units are not in use in the initial architecture implementation [8]. As described in Chapter 4, the non-volatile nature of the on-chip flash memory cells allows the functional unit to be switched off while retaining the configuration in the powered-off state. The power estimation analysis done in this section makes certain assumptions regarding the embedded flash memory implementation on the processor architecture.

The valid assumptions considered in the power estimations are discussed in section 5.3.1. In section 5.3.2, an example application from the application suite is chosen to show the mapping of the power states onto the application execution schedule. The power savings obtained in all of the functional units, and the power and energy savings obtained across the application suite on representative architectures are graphically represented in Section 5.3.3. The power consumption results supporting the graphical representations in this chapter are given in Appendix D.

5.3.1 Assumptions in Power Estimation

It is assumed that the CMOS static-RAM register cells are replaced by the flash memory cells in the wrapper configuration modules of the functional unit. The flash memory cells have slower access rate as compared to the static-RAM cells. For example, the 256K low power SRAM from Hitachi has an access time of 50ns, whereas 256K NOR flash memory cell from Toshiba offers an access rate of 80ns [50]. However, as the clock speed of both the SRAM and flash memory cells is faster the clock rate of the targeted applications, therefore, it is assumed that the access rate does not have a significant affect in this case.

The area of a single flash memory cell depends on the type of the flash memory (NAND or NOR). A single NOR flash memory cell has an area of $0.24\ \mu m^2$ as compared to $0.296\ \mu m^2$ area of CMOS static-RAM cell on 45nm cell technology [51, 52]. Thus, it is safe to assume that the use of flash memory cells does not negatively affect the total area of the architecture.

The energy consumption due to write operations to flash memory is not significant, because all flash write operations are restricted to configuration data written prior to the execution. The output bus-interface reads the flash cells a large number of times during an application execution. The energy consumption of a flash cell per read operation is $0.81\ nJ$ as compared to $0.2\ nJ$ in SRAM cell [52][53]. The estimation of the energy overhead caused due to the read operations is beyond the scope of this thesis and needs to be explored in the future.

A power manager circuitry is typically used to manage the power within the architecture. The approximate power consumption due to the power manager circuitry is $10.1\ \mu W$ [8]. It is not significant in complex applications that heavily reuse the functional units in the architecture [8]. Therefore, this additional power-penalty is not taken into consideration for the power analysis of flash based architecture.

5.3.2 Power States of a Functional Unit

A functional unit has four primary modules, namely, an input bus-interface, a wrapper configuration, an internal module, and an output bus-interface. Each module is powered-on before its execution and powered-down after the execution is complete. Based on the application execution, there are eight feasible power states of a functional unit in the flash-based architecture as shown in Table 4.1.

The execution of Application-III on Architecture-III is considered for the high-level illustration of power states of functional units of an architecture in a single schedule period, as shown in Figure 5.3. From Figure 5.3 it is clear that, the total application schedule Application-II is 28 cycles (active cycles), in which all the functional units operate in an event-driven manner. There are 47 idle cycles during which there is no activity on the bus. The number of active and idle cycles depend on the application clock frequency, which is set to 1000 KHz for this experimental setup. $\text{OFF}_{complete}$ (OFF,OFF,OFF,OFF) is the power state during the idle cycles in which all the modules of a functional unit are powered-down. As Figure 5.3 shows, the powered-down state consumes the majority of the execution cycles in a schedule period, and hence responsible for significant power reduction. Also, during the busy (active) cycles, the functional units are completely switched off after their execution. This leads to power savings in the busy cycles as well. It is also evident from Figure 5.3 that, the individual modules of the functional units, after their respective execution, are powered-down, further adding to the power savings. The FIFO differs from the other functional units in terms of its template and functioning. The power states of the FIFO are given in Appendix D.4. The power savings achieved with respect to the non-flash-based architecture is presented in the next section.

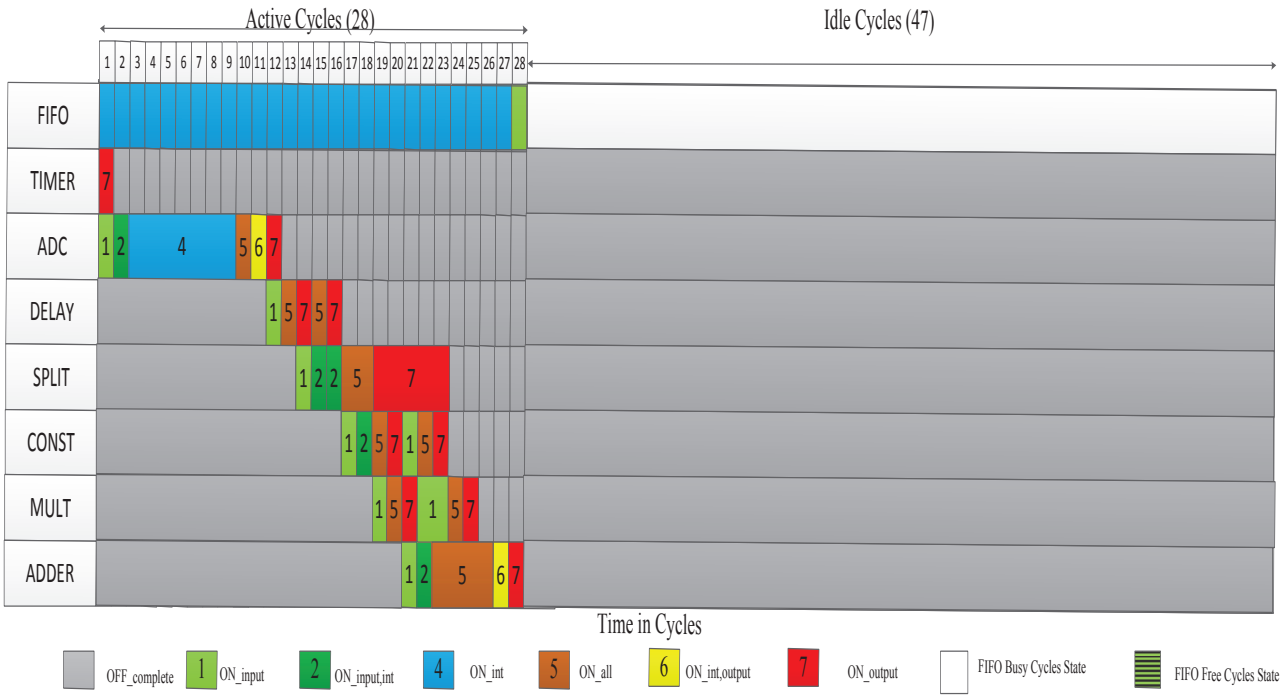

Figure 5.3: Power States of Functional Units in a Schedule Period

5.3.3 Power Savings in Embedded Flash-based Architecture

The power savings obtained in the flash-based architecture with respect to the non-flash-based architecture for each functional unit of Architecture-III are shown in this section. Power dissipated in the architecture design is divided into two categories: *static* and *dynamic*. Static power is the power consumed by the circuit when it is not switching. Dynamic power is the power dissipated when the circuit is active i.e. performing some function. Dynamic power is further divided into two components: *switching power* and *internal power*. The details of the power consumption in each power state of both non-flash-based and flash-based architecture are described in Appendix D.1 and D.2, respectively. Figure 5.4 gives an overview of percentage average power savings in all the functional units of the architecture-III for the execution of Application-III at 500 *KHz* application frequency.

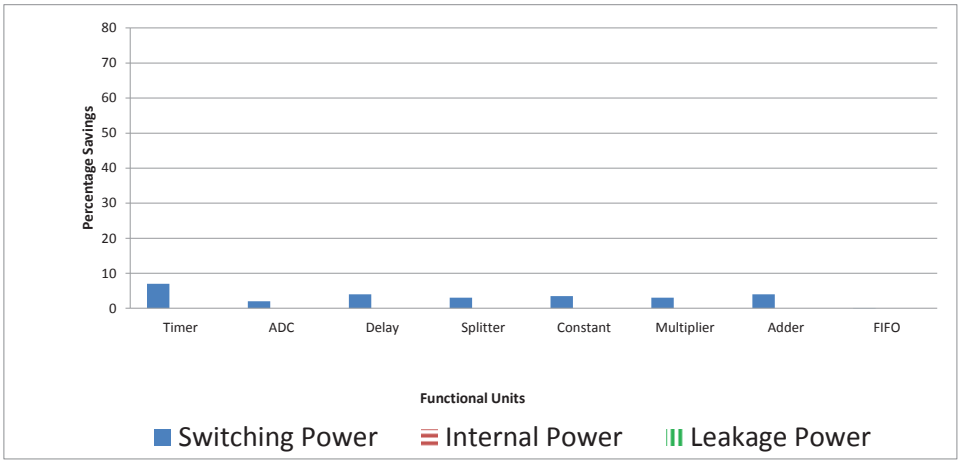

Figure 5.4: Functional Unit Power Savings with Respect to Non-flash-based Architecture

From Figure 5.4, there is a significant reduction in the leakage power dissipation in all the functional units of the flash-based architecture. This is because, there is no leakage current in the idle states. Also, the ratio of the idle cycles to the active cycles is high for this particular application execution (122/28). The powered-down state of the functional units during the idle cycles is primarily responsible for the internal power reduction.

However, the use of flash memory cells does not have a significant effect on the switching power consumption. This is because when a functional unit is in idle state, there is no switching activity in its modules. The power consumption of the input-FIFO in the non-flash based architecture is 42.6*(μW)*, with the leakage power consumption of its internal unit being a dominant factor in it (55%). Some portions of the input-FIFO are powered down during the cycles when there is no data transfer between the network and the processor architecture. This brings the total power consumption of input-FIFO down by 55.45%. Appendix D.4 provides the actual power consumption results of all the functional units of Architecture-III.

5.3.4 Power and Energy Savings Across the Application Suite

The average power and energy savings in the flash-memory-based architecture with respect to the non-flash-based architecture, for a single schedule period of the entire application suite, are presented in Figure 5.5. The energy consumption is calculated using the equation 5.2.

$$E_{sched} = (P1_{avg}) * a * N1 + (P2_{avg}) * a * N2, where, \tag{5.2}$$

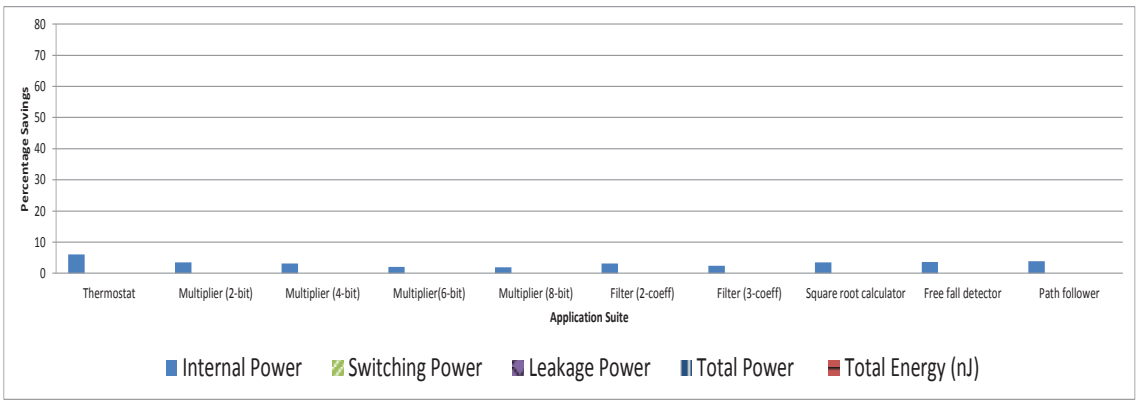

Figure 5.5: Power Savings Across the Application Suite

E_{sched} is the energy consumed in a single schedule period in Joules,

$(P1_{avg})$ is the average power consumption for a busy cycle in Watts,

$(P2_{avg})$ is the average power consumption for a free cycle in Watts,

a is the processor clock period in seconds,

$N1$ is the number of busy cycles in a single schedule period, and

$N2$ is the number of free cycles in a single schedule period.

The number of busy and idle cycles and the power and energy consumptions in the flash-based architecture and the non-flash based architecture for the application suite are presented in Table 5.8.

From Figure 5.5, it is evident that the percentage power savings with respect to the non-flash-based architecture are not uniform for all the applications. The idle cycles in a schedule

Table 5.8: Avg Power, Energy Consumption of Applications in a Schedule Period

App Suite	Active Cycles	Idle Cycles	Flash Power/Non-Flash Power (μW)				Energy (nJ)
			Switching	Internal	Leakage	Total	
I	26	124	2.9/3.1	8.1/22.5	46.0/158.2	57.0/183.9	3.1/11.0
II (2-bit)	46	104	5.5/5.7	13.5/32.2	75.6/198.8	94.6/236.7	8.7/31.0
II (4-bit)	65	85	6.6/7.4	20.8/44.3	114.3/237.8	141.8/289.5	14.8/37.1
II (6-bit)	93	57	11.5/11.5	43.2/70.9	198.6/339.3	253/421.7	47.0/74.6
II (8-bit)	126	24	16.3/16.4	83.2/104.7	296.4/413.8	395.8/534.9	99.7/131.2
III (2-co)	28	122	3.8/3.9	7.3/24.4	48.6/173.1	67.8/201.4	3.8/11.1
III (3-co)	34	116	4.1/4.1	8.9/26.4	52.4/181.3	75.9/211.8	5.1/14.0
IV	54	96	6.2/6.3	17.8/37.2	91.1/218.5	115.1/262	12.4/27.0
V	51	99	5.8/6.1	14.7/36.0	89.8/209	110.4/251.1	11.2/23.9
VI	49	101	5.7/5.93	14.0/32.7	82.3/204.5	102.1/243.2	10.0/30.5

period (during which entire functional units are powered-down) are primarily responsible for the reduction in power consumption. As the complexity of an application increases in terms of the execution cycles, the number of idle cycles decreases, thereby reducing the effect of the flash memory cells. Application-II(*8*-bit multiplier), with the least number of idle cycles in the application suite, has a net power saving of 26% in an embedded flash memory-based architecture, whereas Application-I (temperature controller), with the most number of idle cycles, has a net power saving of 69%. Thus, based on the estimations of the power-savings achieved, this thesis proposes the use of embedded flash memory for reduction in the power consumption.

Chapter 6

Conclusion

This thesis demonstrates the design of an MDE-based application framework for the rapid generation of software applications to explore the processor architecture. Simulink, a model-based design tool, is used to develop applications in its graphical environment. The application framework provides a simple application programming interface by successfully generating application graphs from the Simulink block diagram. Also, the architecture design is made flexible to allow execution of simple as well as complex applications. A wide range of applications of varying complexity are successully executed on the architecture, validating flexibility of the architecture design. Additionally, a detailed power analysis is done to estimate the power savings achieved by the embedded flash memory in the architecture. The advantages of incorporation of flash memory are demonstrated by comparing its power consumption with that of non-flash based architecture. It is shown that the embedded flash memory based causes significant power savings.

The work done in this thesis combined with the work presented in [8], [6], and [35] demonstrate the design of a flexible dataflow architecture suitable for event-driven applications. It also presents a robust MDE-based automation framework facilitating easier application de-

vevlopment, automatic configuration generation, and validation of the application execution on the architecture.

6.1 Future Work

The application framework developed in this thesis makes use of Simulink as an application modeling platform. The current implementation supports the predefined blocks, subsystems, and the embedded matlab functions for the application development. This implementation can be expanded further to encompass other libraries of the Simulink (for example, State-Flow, SimEvents) to enhance the application generation flexibility. Also, application clock information is provided externally in the current implementation of the framework. The Simulink clock can be tuned to the archicteture clock to complete the automation of the test-bench generation. The new architecture implementation consists of a single input-FIFO and a single output-FIFO for the communication with the network. It can be explored further to include multiple FIFOs. The power estimation of the embedded flash memory based architecture does not take into consideration the power penalty caused due to the memory write operations. A qualtitative analysis of the impact of the power penalty is required in the future.

Bibliography

[1] D. Marculescu, R. Marculescu, Zamora, Stanley-Marbell, K. Park, J. Jung, L. Weber, K. Cottet, Grzyb, Troster, Jones, Martin, and Nakad, "Electronic textiles: a platform for pervasive computing," *Proceedings of the IEEE*, vol. 91, pp. 1993 – 1994, dec 2003.

[2] D. Marculescu, "E-textiles: toward computational clothing," *Pervasive Computing, IEEE*, vol. 2, pp. 89 – 95, jan-mar 2003.

[3] Josh Edmison, Mark Jones, Thurmon Lockhart, Thomas Martin, *Wearable eHealth Systems for Personalised Health Management: State of the Art and Future Challenges*, ch. An e-Textile System for Motion Analysis, pp. 292–301. IOS Press, 2005.

[4] T. M. M. Jones and Z. Nakad, "A service backplane for e-textile applications," 2002. Workshop on Modeling, Analysis, and Middleware Support for Electronic Textiles.

[5] J.-C. Kao and R. Marculescu, "Energy-aware routing for e-textile applications," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, (Washington, DC, USA), pp. 184–189, IEEE Computer Society, 2005.

[6] K. Lakshmanan, "Design of an automation framework for a novel data-flow processor architecture," 2010. Masters thesis, Virginia Polytechnic Institute and State University.

[7] "Vt e-textiles group." `http://www.ccm.ece.vt.edu/etextiles/`.

[8] R. Narayanswamy, "Design of a power-aware dataflow processor architecture," 2010. Masters thesis, Virginia Polytechnic Institute and State University.

[9] "Simulink and model-based design." `http://www.mathworks.com/products/simulink/`.

[10] D. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, pp. 25 – 31, feb. 2006.

[11] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema, "Developing applications using model-driven design environments," *Computer*, vol. 39, pp. 33 – 40, feb. 2006.

[12] "Simulink embedded code generation." http://www.mathworks.com/embedded-code-generation/index.html.

[13] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, pp. 127 – 144, jan 2003.

[14] "Reactis: Model-based design with simulink and stateflow." http://www.reactive-systems.com/.

[15] "Stateflow: Design and simulate state charts." http://www.mathworks.com/products/stateflow/.

[16] "Unified modeling language (uml)." http://www.omg.org/spec/UML/.

[17] "The uml profile for marte: Modeling and analysis of real-time and embedded systems." http://www.omgmarte.org/.

[18] "Alderis@uci." http://alderis.ics.uci.edu/.

[19] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo, "Peace: A hardware-software codesign environment for multimedia embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, pp. 24:1–24:25, May 2008.

[20] S. Ha, "Model-based programming environment of embedded software for mpsoc," in *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, pp. 330 –335, jan. 2007.

[21] D. de Niz, G. Bhatia, and R. Rajkumar, "Model-based development of embedded systems: The sysweaver approach," in *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pp. 231 – 242, april 2006.

[22] G. Madl, S. Pasricha, N. Dutt, and S. Abdelwahed, "Cross-abstraction functional verification and performance analysis of chip multiprocessor designs," *Industrial Informatics, IEEE Transactions on*, vol. 5, pp. 241 –256, aug. 2009.

[23] J. Dekeyser, P. Boulet, P. Marquet, and S. Meftali, "Model driven engineering for soc co-design," in *IEEE-NEWCAS Conference, 2005. The 3rd International*, pp. 21 – 25, june 2005.

[24] M. Hempstead, N. Tripathi, P. Mauro, G.-Y. Wei, and D. Brooks, "An ultra low power system architecture for sensor network applications," *SIGARCH Comput. Archit. News*, vol. 33, pp. 208–219, May 2005.

[25] M. Seok, S. Hanson, Y.-S. Lin, Z. Foo, D. Kim, Y. Lee, N. Liu, D. Sylvester, and D. Blaauw, "The phoenix processor: A 30pw platform for sensor applications," in *VLSI Circuits, 2008 IEEE Symposium on*, pp. 188 –189, june 2008.

[26] M. Hempstead, G.-Y. Wei, and D. Brooks, "Architecture and circuit techniques for low-throughput, energy-constrained systems across technology generations," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, (New York, NY, USA), pp. 368–378, ACM, 2006.

[27] S. M. Martin, K. Flautner, T. Mudge, and D. Blaauw, "Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads," in *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, (New York, NY, USA), pp. 721–725, ACM, 2002.

[28] "Unified power format (upf) standard." `http://www.unifiedpowerformat.com/images/UPF.v1.0Standard.pdf`.

[29] J. Brewer and M. Gill, "Nonvolatile memory technologies with emphasis on flash(a comprehensive guide to understanding and using flash memory devices)," p. 750, 2008.

[30] H. Hidaka, "Evolution of embedded flash memory technology for mcu," in *IC Design Technology (ICICDT), 2011 IEEE International Conference on*, pp. 1 –4, may 2011.

[31] Y. Shin, J. Choi, C. Kang, C. Lee, K.-T. Park, J.-S. Lee, J. Sel, V. Kim, B. Choi, J. Sim, D. Kim, H. ju Cho, and K. Kim, "A novel nand-type monos memory using 63nm process technology for multi-gigabit flash eeproms," in *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*, pp. 327 –330, dec. 2005.

[32] B. Eitan, P. Pavan, I. Bloom, E. Aloni, A. Frommer, and D. Finzi, "Nrom: A novel localized trapping, 2-bit nonvolatile memory cell," *Electron Device Letters, IEEE*, vol. 21, pp. 543 –545, nov 2000.

[33] E. Lusky, Y. Shacham-Diamand, I. Bloom, and B. Eitan, "Electrons retention model for localized charge in oxide-nitride-oxide (ono) dielectric," *Electron Device Letters, IEEE*, vol. 23, pp. 556 – 558, sep 2002.

[34] S.-R. Kim, K. J. Han, K.-S. Lee, R. Li, J. Wolfman, T.-H. Kim, P. Liu, H. Kim, P.-Y. Lee, Y. Wang, Y. Jia, F. Dhaoui, F. Hawley, and H.-C. Tseng, "High performance 65nm 2t-embedded flash memory for high reliability soc applications," in *Memory Workshop (IMW), 2010 IEEE International*, pp. 1 –3, may 2010.

[35] S. Malayattil, "Design of a multi-bus data-flow processor architecture," 2012. (to be published).

[36] "Simulink subsystems." `http://www.mathworks.com/help/toolbox/simulink/ug/f4-53172.html`.

[37] "Simulink matlab function." `http://www.mathworks.com/help/toolbox/simulink/ug/f6-6010.html`.

[38] "Simulink library." http://conqat.cs.tum.edu/index.php/ConQAT.

[39] "Synopsys low power design tools," 2007. https://electronics.wesrch.com/Userimages/Pdf/SE11191538495.pdf,2007.

[40] "Tsmc 45nm design ecosystem in place," 2010. http://www.magma-da.com/partners/StandardCellLibs_45nm.aspx.

[41] P. Pavan, R. Bez, P. Olivo, and E. Zanoni, "Flash memory cells-an overview," *Proceedings of the IEEE*, vol. 85, pp. 1248 –1271, aug 1997.

[42] "Thermostat reference design using the mc9s08ll16," 2010. http://www.freescale.com/files/microcontrollers/doc/ref_manual/DRM106.pdf.

[43] "Pic16c5x / pic16cxxx math utility routines," 2010. http://ww1.microchip.com/downloads/en/AppNotes/00526e.pdf.

[44] "Avr223: Digital filters with av," 2008. http://www.atmel.com/dyn/resources/prod_documents/doc2527.pdf.

[45] "Fast integer square root," 2000. http://ww1.microchip.com/downloads/en/AppNotes/91040a.pdf,.

[46] R. G. Lyons, "Understanding digital signal processing, secondedition," p. 358, 2004.

[47] "Measuring freefall using freescales mma7360l 3-axis acceleromete," 2007. http://ww1.microchip.com/downloads/en/AppNotes/91040a.pdf,.

[48] "Lis331a - 3-axis accelerometer datasheet," 2007. http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATASHEET/CD00172239.pdf.

[49] "Synopsys 90nm generic library for teaching ic design." http://www.synopsys.com/community/universityprogram/pages/library.aspx.

[50] "Sram technologys." smithsonianchips.si.edu/ice/cd/MEMORY97/SEC8.PDF.

[51] F.-L. Yang, C.-C. Huang, C.-C. Huang, T.-X. Chung, H.-Y. Chen, C.-Y. Chang, H.-W. Chen, D.-H. Lee, S.-D. Liu, K.-H. Chen, C.-K. Wen, S.-M. Cheng, C.-T. Yang, L.-W. Kung, C.-L. Lee, Y.-J. Chou, F.-J. Liang, L.-H. Shiu, J.-W. You, K.-C. Shu, B.-C. Chang, J.-J. Shin, C.-K. Chen, T.-S. Gau, P.-W. Wang, B.-W. Chan, P.-F. Hsu, J.-H. Shieh, S.-H. Fung, C. Diaz, C.-M. Wu, Y.-C. See, B. Lin, M.-S. Liang, J.-C. Sun, and C. Hu, "45nm node planar-soi technology with 0.296 mu;m2 6t-sram cell," in *VLSI Technology, 2004. Digest of Technical Papers. 2004 Symposium on*, pp. 8 – 9, june 2004.

[52] R. Fastow, R. Banerjee, P. Bjeletich, A. Brand, H. Chao, J. Gorman, X. Guo, J. Heng, N. Koenigsfeld, S. Ma, A. Masad, S. Soss, and B. Woo, "A 45nm nor flash technology with self-aligned contacts and 0.024 um2 cell size for multi-level applications," in *VLSI Technology, Systems and Applications, 2008. VLSI-TSA 2008. International Symposium on*, pp. 81 –82, april 2008.

[53] A. Azizi-mazreah, M. T. M. Shalmani, H. Barati, and A. Barati, "Delay and energy consumption analysis of conventional sram," 2008.

Appendix A

Simulink Application Suite

Simulink models of the application suite described in Section 5.1 is presented in this chapter.

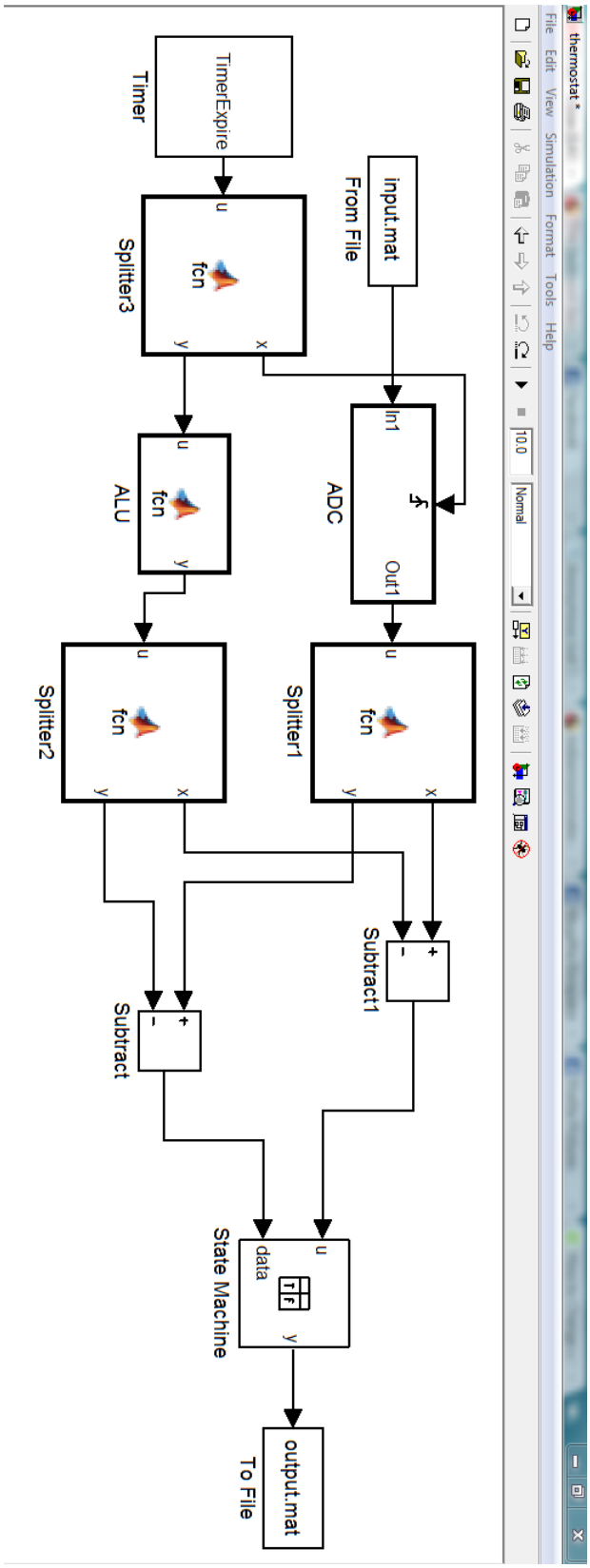

Figure A.1: Simulink Temperature Controller Model

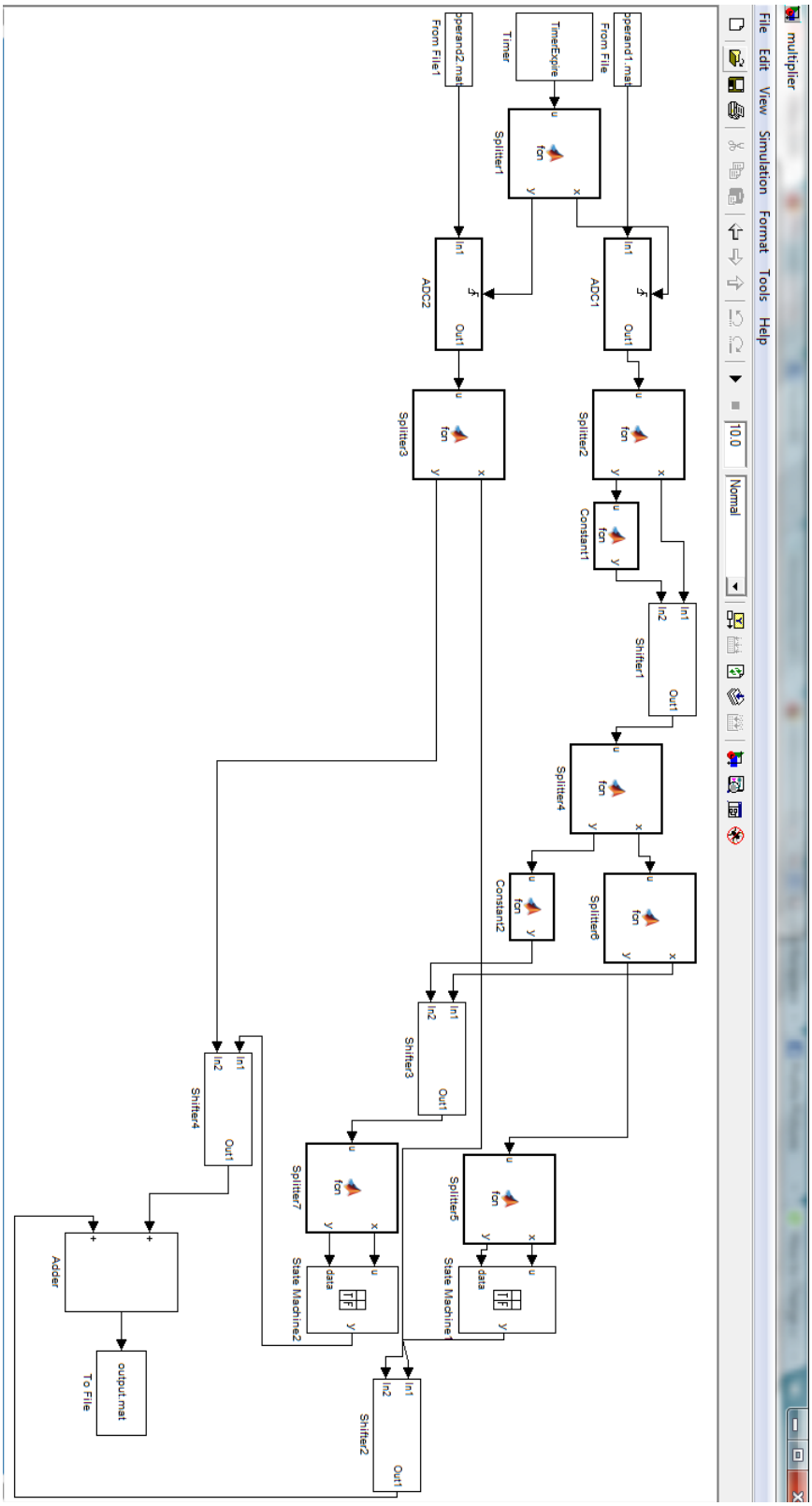

Figure A.2: Simulink Multiplier Model

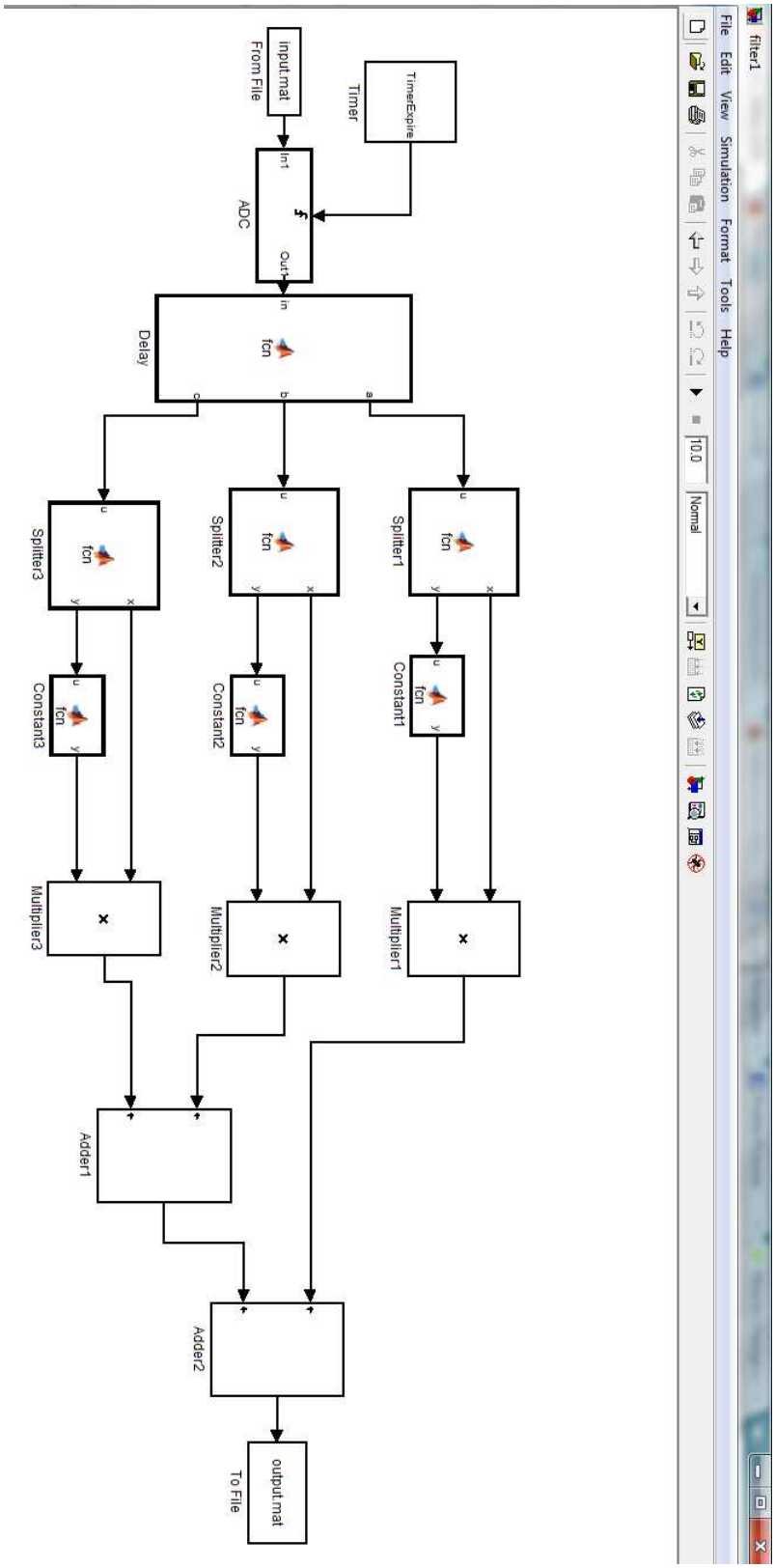

Figure A.3: Simulink Filter Model

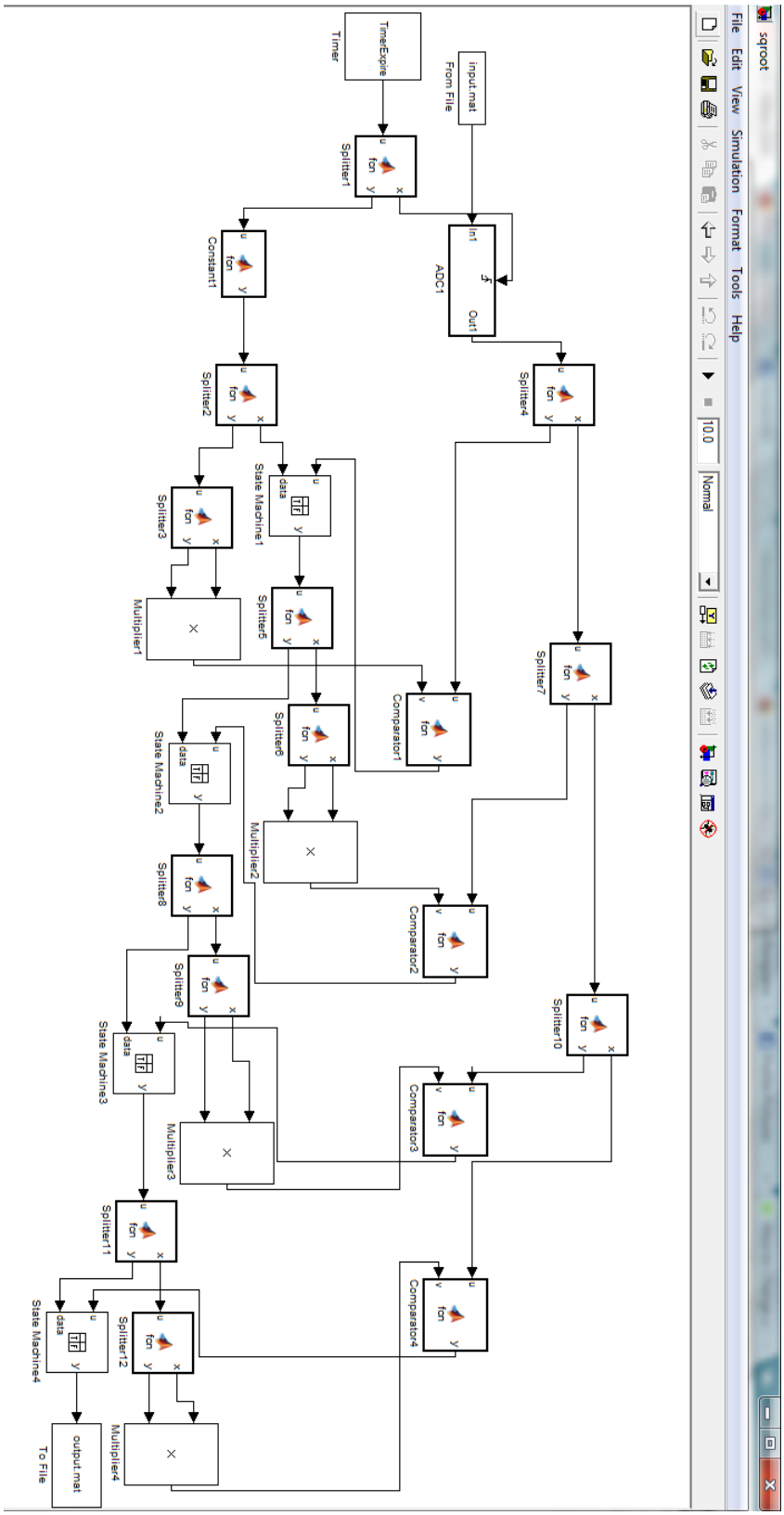

Figure A.4: Simulink Square Root Model

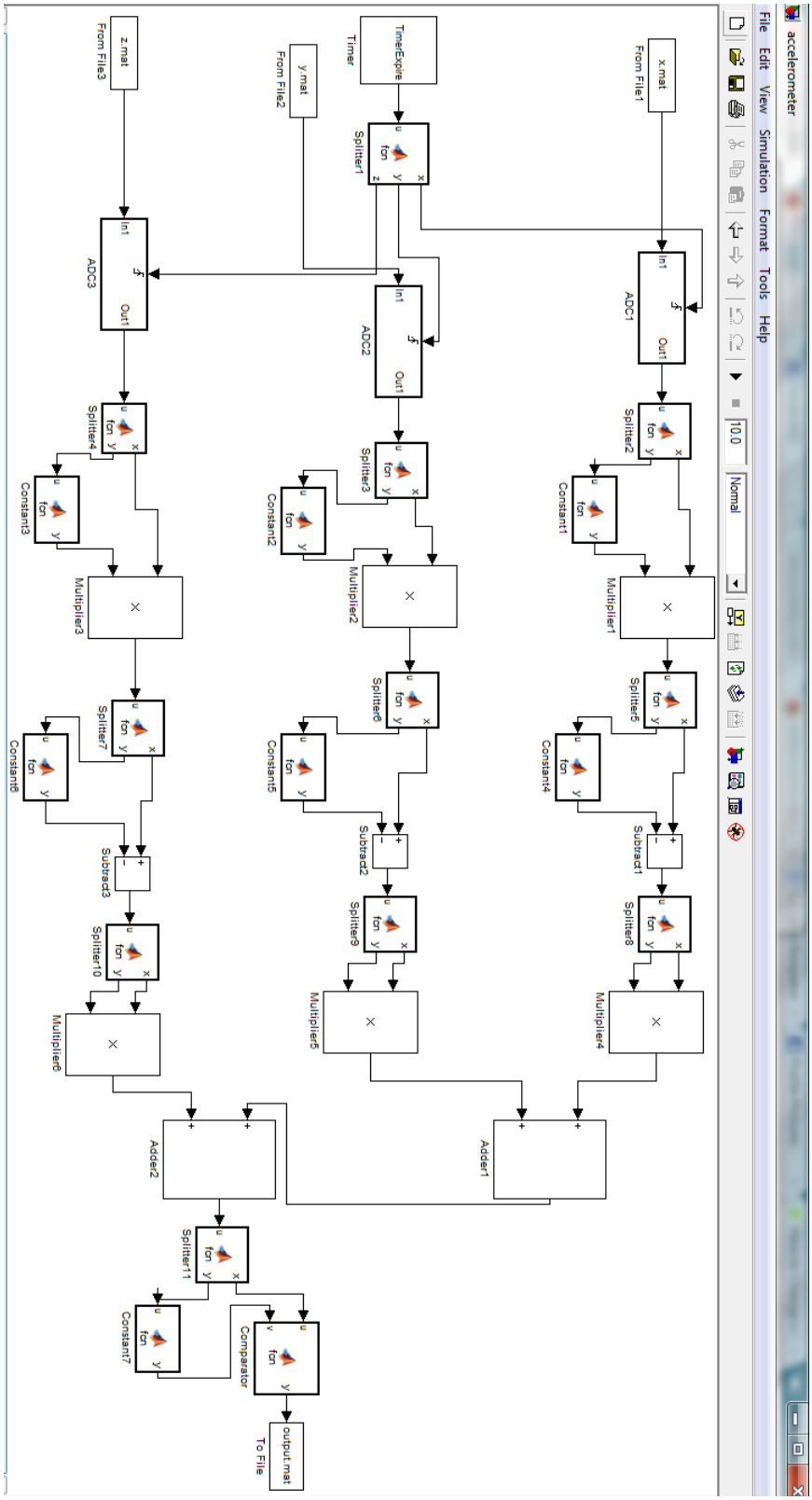

Figure A.5: Simulink Free Fall Detector Model

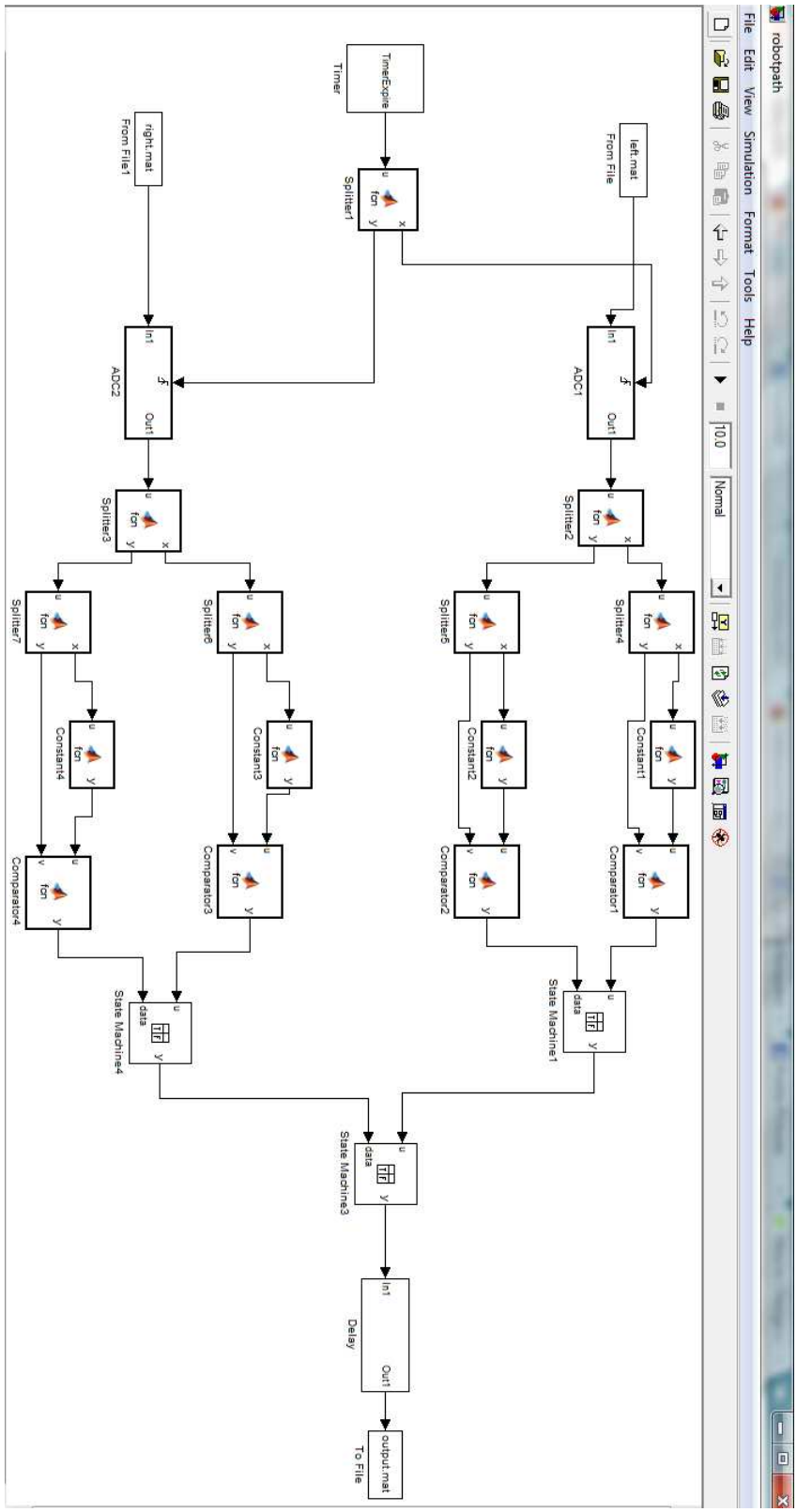

Figure A.6: Simulink Path Follower Model

Appendix B

Application Graphs

Application graphs generated from simulink models of the application suite are shown in this chapter.

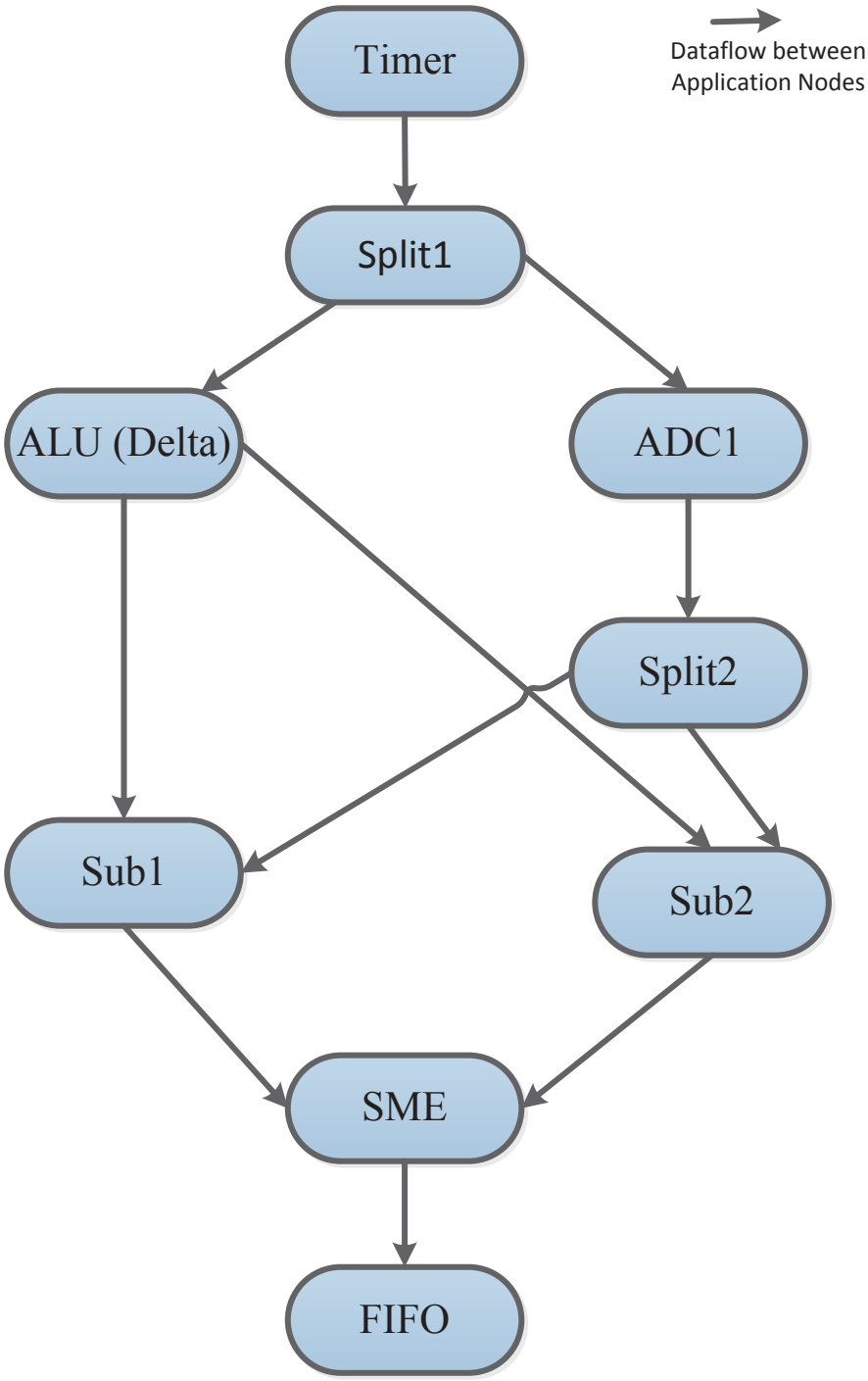

Figure B.1: Temperature Controller Application Graph

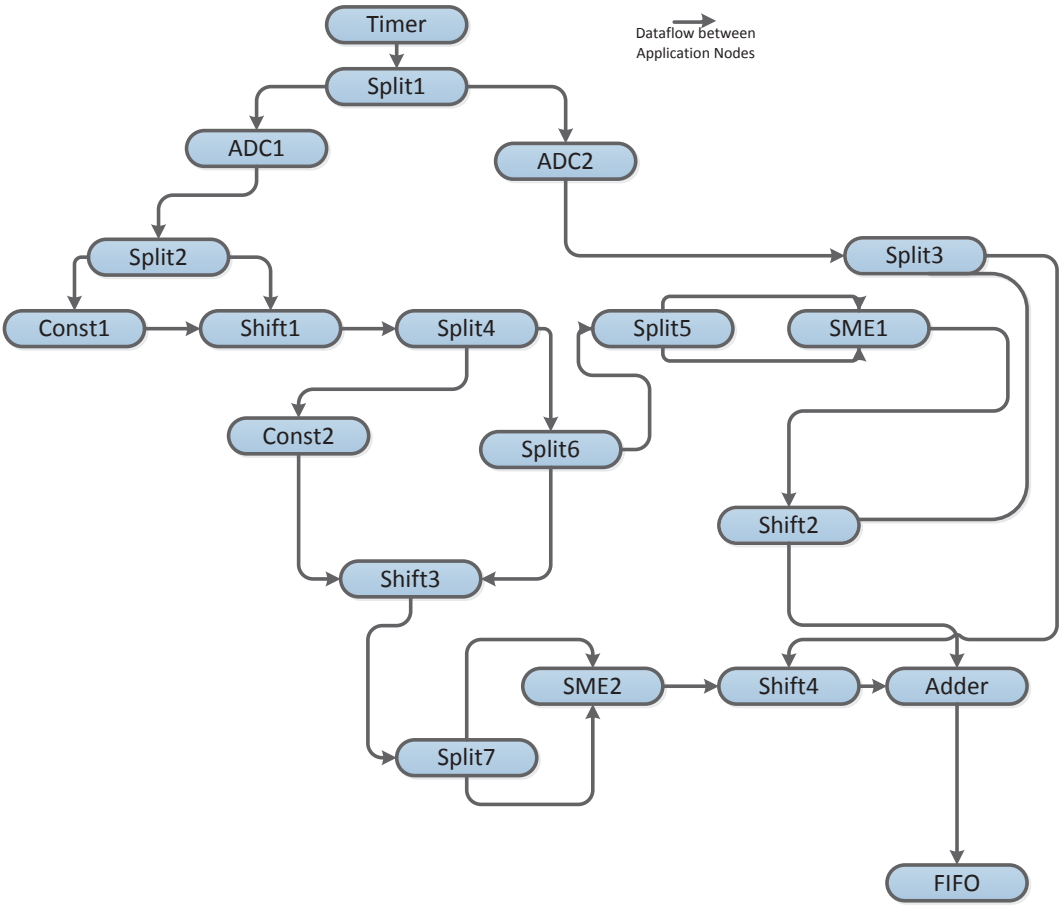

Figure B.2: Application-II

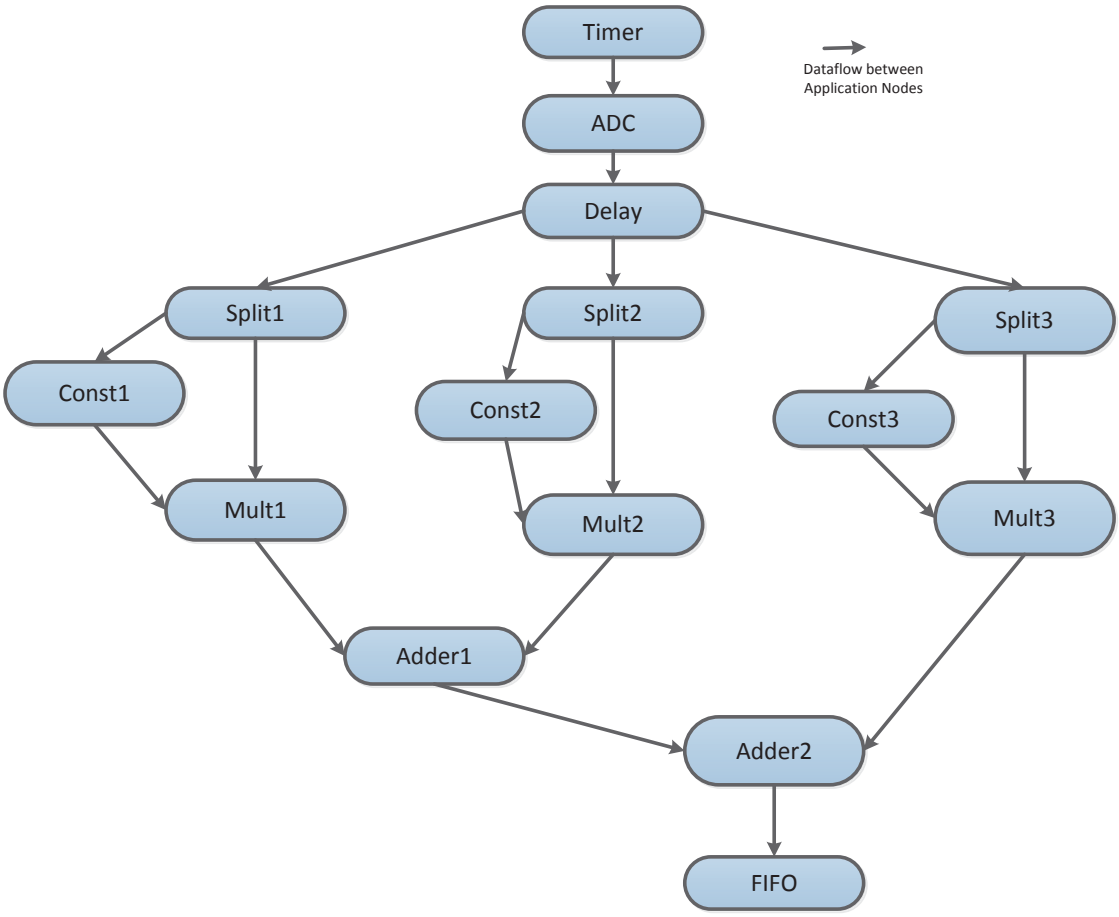

Figure B.3: Application-III (3-coefficient) Graph

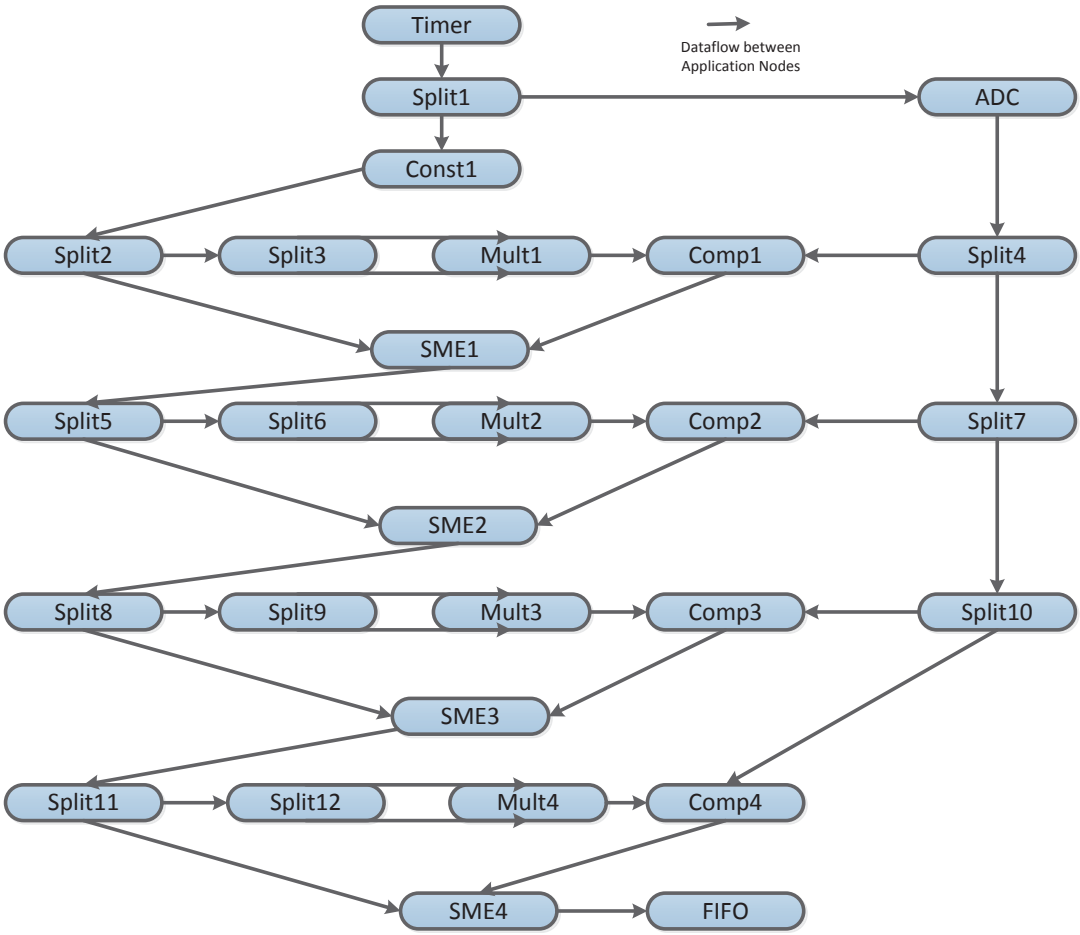

Figure B.4: Application-V Graph

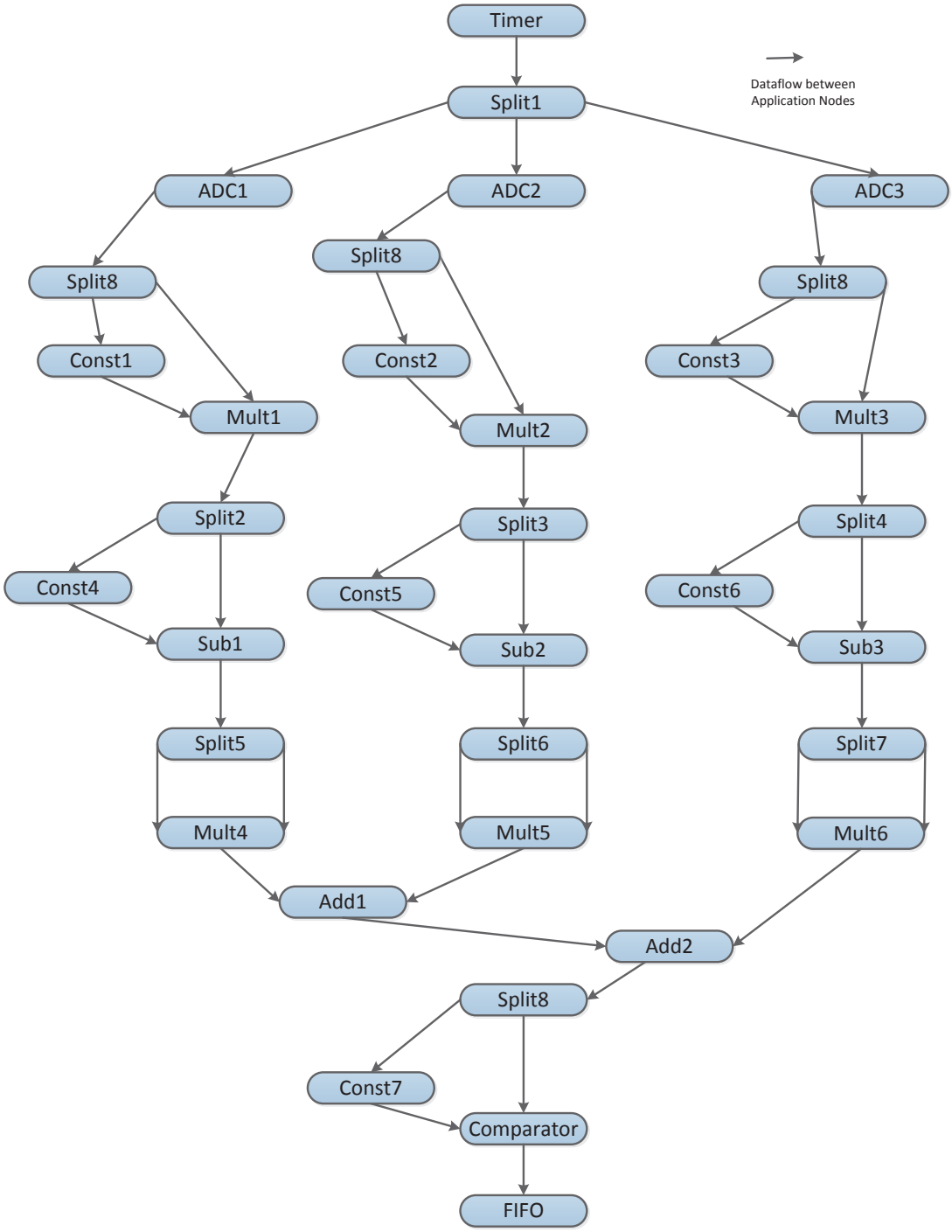

Figure B.5: Application-IV Graph

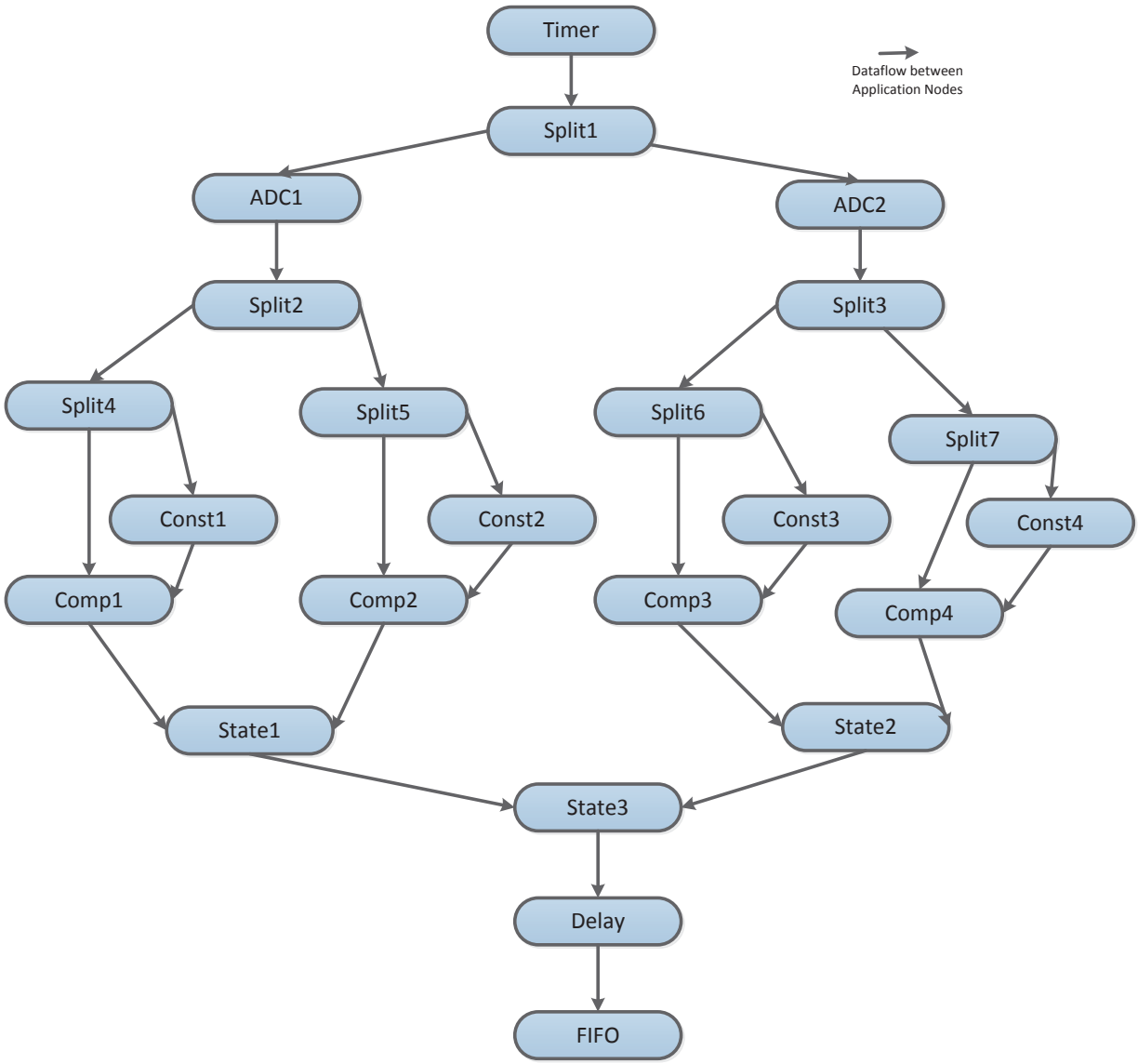

Figure B.6: Application-VI Graph

Appendix C

Architecture Sets

The graphical representations of the architecture set described in Section 5.3 is shown in this chapter.

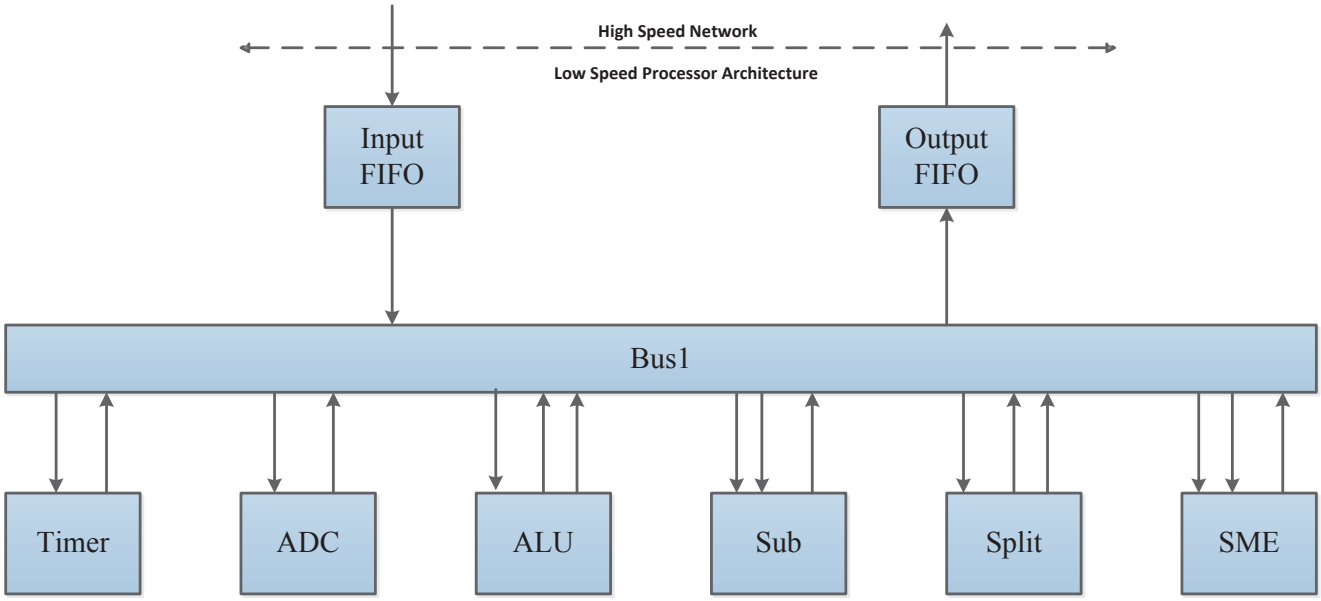

Figure C.1: Architecture-I

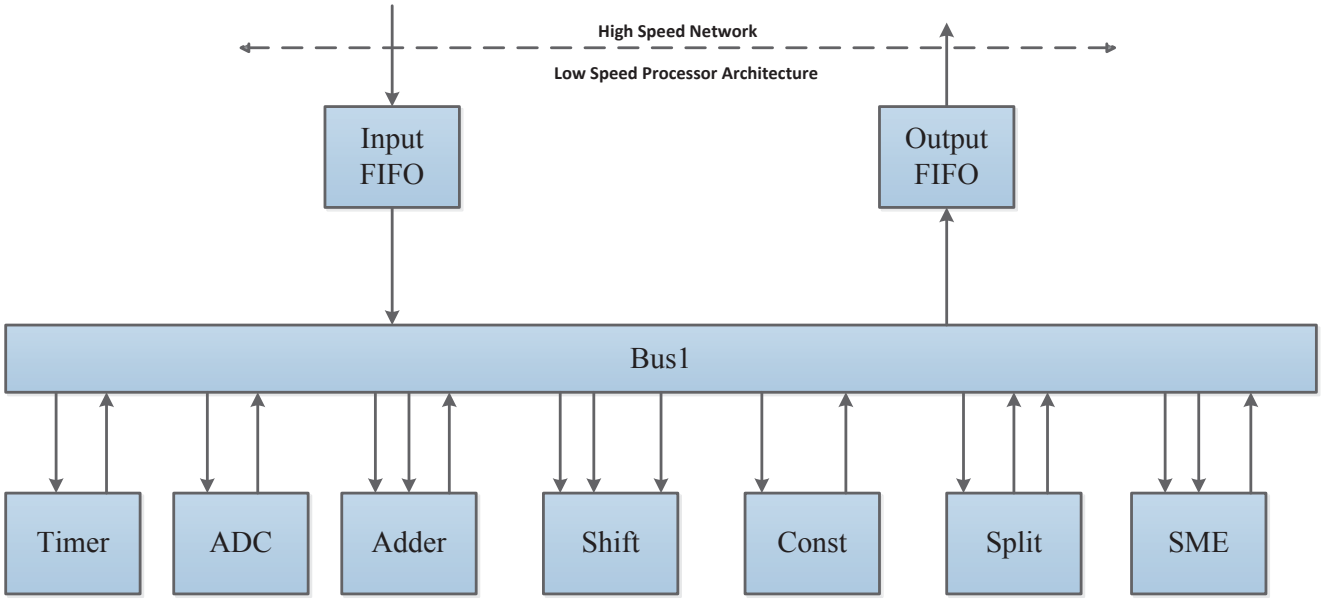

Figure C.2: Architecture-II

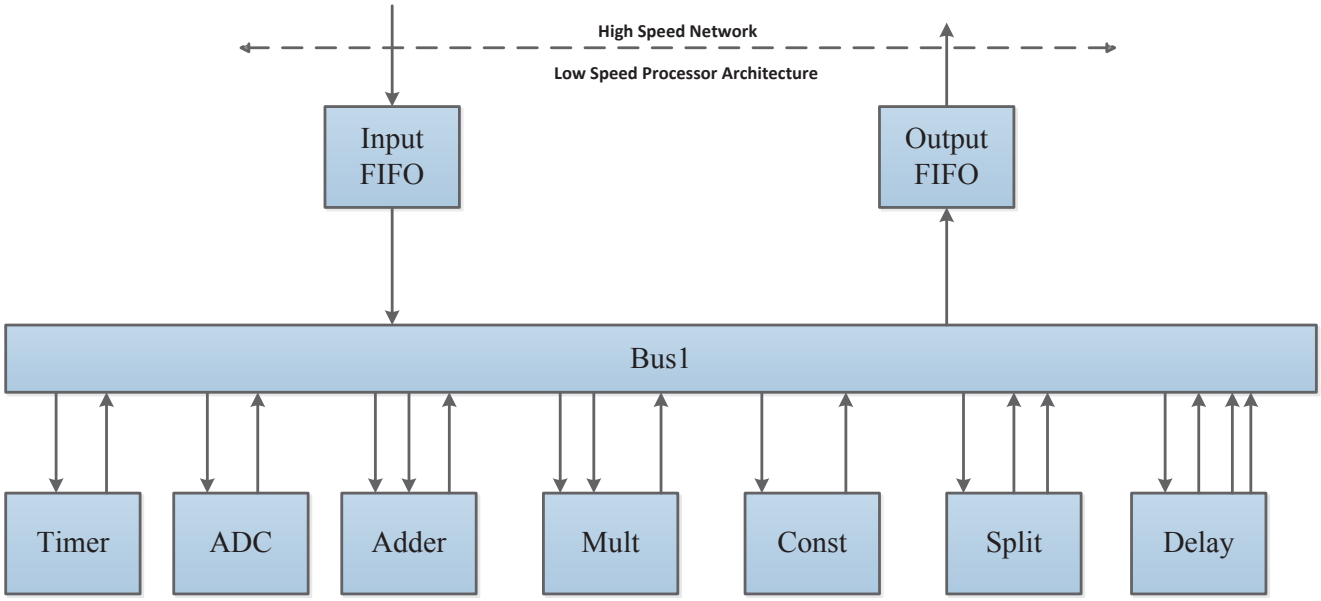

Figure C.3: Architecture-III

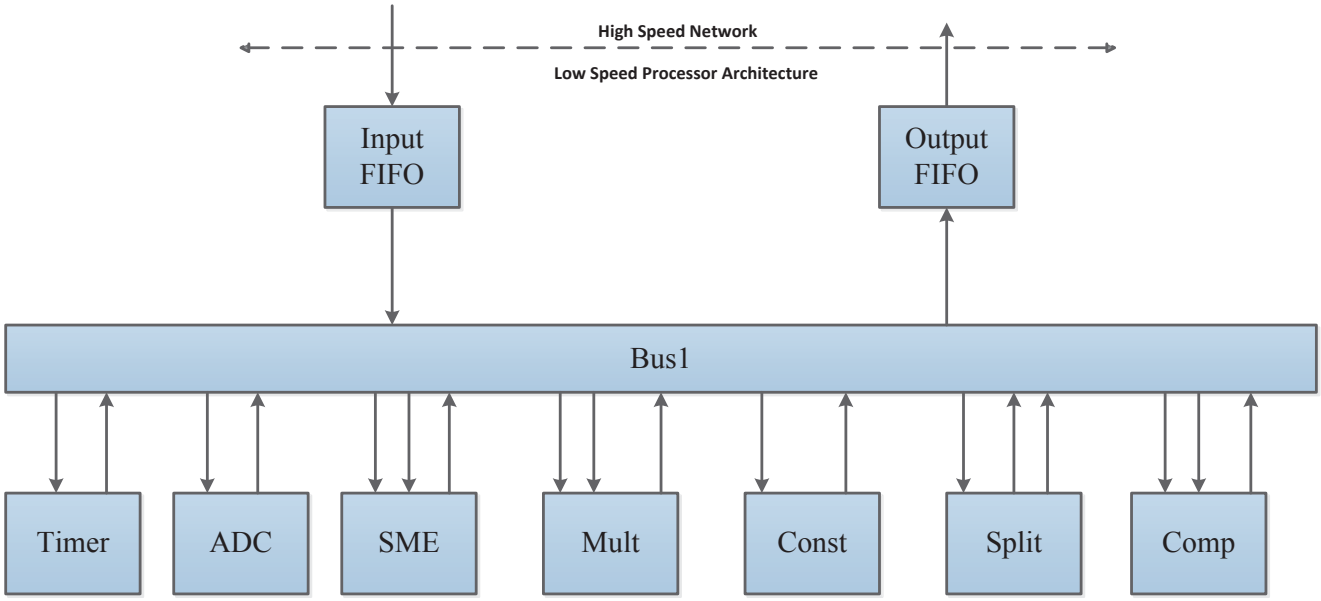

Figure C.4: Architecture-IV

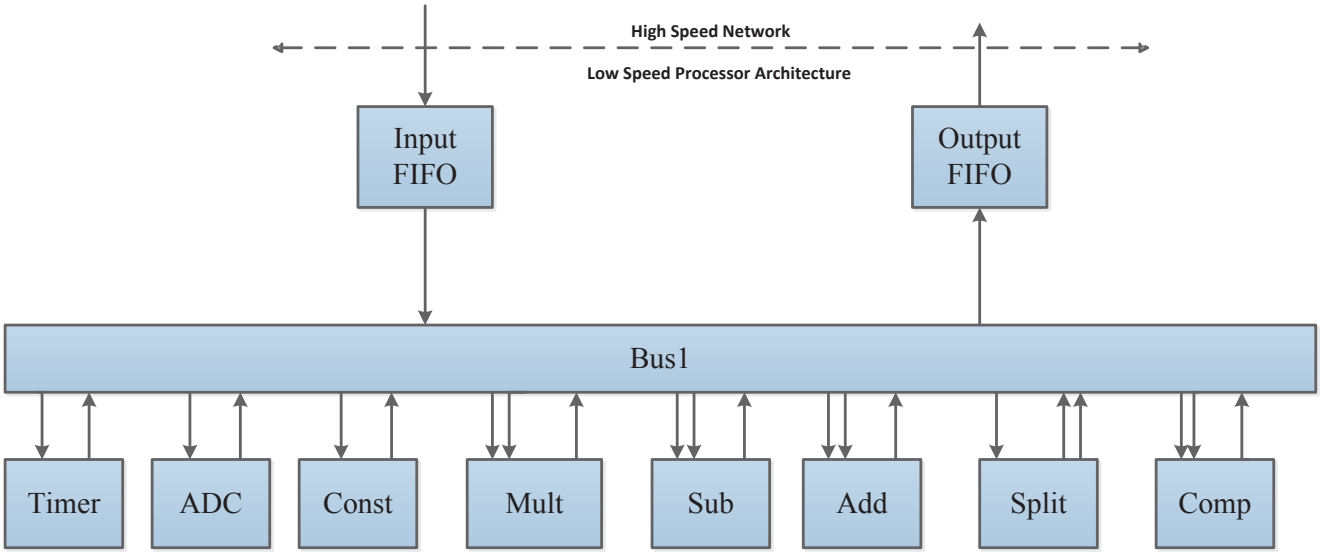

Figure C.5: Architecture-V

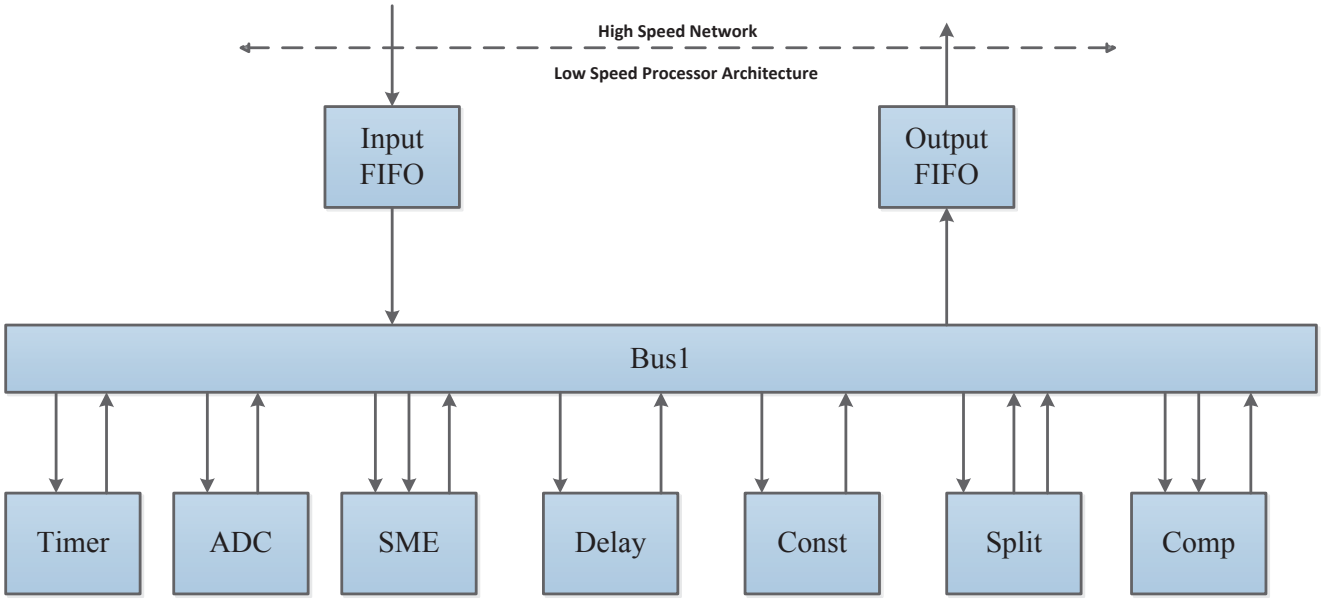

Figure C.6: Architecture-VI

Appendix D

Power Results

Detailed power consumption values supporting the power savings graphs in Section 5.3.2 are provided in this section.

D.1 Power Consumption in Non-flash-based Architecture

All types of power dissipation in each feasible power state of a non-flash-based architecture is given in this section.

Table D.1: Number of Cycles of Power States

Cycle Type	Power States (input, wrapper, internal, output)	Number of Cycles (75)		
		Adder	A/D	Splitter
Free Cycles	(OFF, ON, OFF, OFF)	64	61	63
Busy Cycles	(ON, ON, OFF, OFF)	6	2	2
	(ON, ON, ON, OFF)	1	1	2
	(OFF, ON, ON, OFF)	1	8	1
	(OFF, ON, ON, ON)	2	2	2
	(OFF, ON, OFF, ON)	1	1	5

Table D.2: Leakage Power Consumption in Different Power States

Power States (input, wrapper, internal, output)	Leakage Power *(μW)*		
	Adder	A/D	Splitter
OFF_{input}, $\text{ON}_{wrapper}$, $\text{OFF}_{internal}$, OFF_{output}	11.41	11.34	14.95
ON_{input}, $\text{ON}_{wrapper}$, $\text{OFF}_{internal}$, OFF_{output}	18.87	15.14	18.75
ON_{input}, $\text{ON}_{wrapper}$, $\text{ON}_{internal}$, OFF_{output}	22.78	19.37	23.37
OFF_{input}, $\text{ON}_{wrapper}$, $\text{ON}_{internal}$, OFF_{output}	15.32	15.57	16.42
OFF_{input}, $\text{ON}_{wrapper}$, $\text{ON}_{internal}$, ON_{output}	23.01	23.72	31.82
OFF_{input}, $\text{ON}_{wrapper}$, $\text{OFF}_{internal}$, ON_{output}	19.10	19.03	30.35

Table D.3: Dynamic Power Consumption in Power States

Power States (input, wrapper, internal, output)	Dynamic Power *(μW)*					
	Adder		A/D		Splitter	
	Int	Switch *(nW)*	Int	Switch *(nW)*	Int	Switch *(nW)*
$\text{OFF}_{i/p}$, ON_{wr}, OFF_{int}, $\text{OFF}_{o/p}$	3.29	0	3.29	0	3.52	0
$\text{ON}_{i/p}$, ON_{wr}, OFF_{int}, $\text{OFF}_{o/p}$	4.15	749	3.72	395	3.95	432
$\text{ON}_{i/p}$, ON_{wr}, ON_{int}, $\text{OFF}_{o/p}$	4.44	997	4.1	410	4.45	1241
$\text{OFF}_{i/p}$, ON_{wr}, ON_{int}, $\text{OFF}_{o/p}$	3.58	248	3.67	515	4.1	809
$\text{OFF}_{i/p}$, ON_{wr}, ON_{int}, $\text{ON}_{o/p}$	4.16	482	4.35	766	5.3	2025
$\text{OFF}_{i/p}$, ON_{wr}, OFF_{int}, $\text{ON}_{o/p}$	4.87	580	3.87	251	4.72	12.16

D.2 Power Consumption in Flash-based Architecture

All types of power dissipation in each feasible power state of a flash-memory-based architecture is given in this section.

Table D.4: Number of Cycles in Different Power States

Cycle Type	Power States (input, wrapper, internal, output)	Number of Cycles (75)		
		Adder	A/D	Splitter
Free Cycles	$OFF_{complete}$	64	61	63
Busy Cycles	(ON, OFF, OFF, OFF)	5	1	1
	(ON, ON, OFF, OFF)	1	1	1
	(ON, OFF, ON, OFF)	1	1	2
	(OFF, OFF, ON, OFF)	1	8	1
	(ON, ON, ON, ON)	1	1	1
	(OFF, OFF, ON, ON)	1	1	1
	(OFF, OFF, OFF, ON)	1	1	5

Table D.5: Leakage Power Consumption in Different Power States

Power States (input, wrapper, internal, output)	Leakage Power (μW)		
	Adder	A/D	Splitter
$OFF_{complete}$	0	0	0
(ON, OFF, OFF, OFF)	15.46	11.80	11.80
(ON, ON, OFF, OFF)	18.87	15.14	18.75
(OFF, ON, ON, OFF)	19.37	16.03	16.42
(OFF, OFF, ON, OFF)	11.91	12.23	12.62
(ON, ON, ON, ON)	23.01	23.72	31.82
(OFF, OFF, ON, ON)	19.60	19.92	28.00
(OFF, OFF, OFF, ON)	15.69	15.69	23.40

Table D.6: Dynamic Power Consumption in Different Power States

Power States (input, wrapper, internal, output)	Dynamic Power *(μW)*					
	Adder		A/D		Splitter	
	Int	Switch *(nW)*	Int	Switch *(nW)*	Int	Switch *(nW)*
$OFF_{complete}$	0	0	0	0	0	0
(ON, OFF, OFF, OFF)	3.86	749	3.43	395	3.43	432
(ON, ON, OFF, OFF)	4.15	749	3.72	395	3.95	432
(OFF, ON, ON, OFF)	3.58	997	3.81	910	4.0	1241
(OFF, OFF, ON, OFF)	3.29	248	3.29	248	3.58	809
(ON, ON, ON, ON)	4.16	482	4.35	766	5.3	2025
(OFF, OFF, ON, ON)	3.87	482	3.96	766	4.78	2025
(OFF, OFF, OFF, ON)	3.58	234	3.58	251	4.2	1216

D.3 Power States of FIFO

All feasible power states of FIFO functional unit are presented in this section.

Table D.7: Power States in Input FIFO

Cycles	Power State of Input FIFO					
	Network interface	wrapper	packet mask	count down	internal	bus interface
Architecture Configuration	ON	OFF	OFF	OFF	OFF	OFF
	ON	OFF	OFF	OFF	ON	OFF
	ON	OFF	OFF	OFF	ON	ON
FIFO Configuration	ON	ON	ON	ON	OFF	ON
Free Schedule Cycles	ON	OFF	ON	ON	ON	ON
Busy Schedule Cycles	ON	OFF	ON	ON	OFF	ON

Table D.8: Power States in Output FIFO

Cycles	Power State of Output FIFO		
	bus-input	internal module	network-nout
Listen Cycles	ON	OFF	OFF
IM Execution Cycles	ON	ON	OFF
Network Transfer cycles	OFF	ON	ON

D.4 Power Estimation of Individual Functional Units

This section presents the comparison between average power consumption of functional units in non-flash-based and flash-based processor architecture.

Table D.9: Average Power Consumption of Functional Units in a Schedule Period

Functional Unit	Dynamic Power (μW)		Leakage Power (μW)		Total Power (μW)	
	w/o Flash	with Flash	w/o Flash	with Flash	w/o Flash	with Flash
Adder	2.43	0.79	12.51	4.36	15.94	5.15
A/D	3.59	0.82	12.16	4.21	15.75	5.03
Splitter	3.10	1.09	25.32	7.64	29.42	8.73
Timer	3.10	0.89	16.42	5.87	19.52	6.76
Multiplier	4.34	1.50	30.95	11.61	35.29	13.11
Constant	2.45	0.83	13.68	4.56	16.13	5.39
Delay	2.58	0.89	23.54	7.07	26.12	7.96
FIFO	7.61	2.15	41.32	13.42	47.93	15.57

D.5 Power Savings Across the Application Suite

This section presents the comparison between average power consumption of entire application suite in non-flash-based and flash-based processor architecture.

Table D.10: Average Power, Energy Consumption of Applications in a Schedule Period

Application Suite	Non-flash Power (μW)					Flash Power (μW)				
	Swt	Int	Lkge	Total	Engy	Swt	Int	Lkge	Total	Engy
I	3.1	22.5	158.2	183.9	11.07	2.9	8.1	46.0	57.0	3.1
II (2-bit)	5.7	32.2	198.8	236.7	31.0	5.5	13.5	75.6	94.6	8.7
II (4-bit)	7.4	44.3	237.8	289.5	37.1	6.6	20.82	114.38	141.8	14.87
II (6-bit)	11.5	70.9	339.3	421.7	74.69	11.2	43.2	198.6	253	47.06
II (8-bit)	16.4	104.7	413.8	534.9	131.23	16.2	83.2	296.4	395.8	99.74
III (2-coeff)	3.97	24.4	173.1	201.47	11.15	3.86	7.36	48.65	67.8	3.8
III (3-coeff)	4.10	26.41	181.33	211.84	14.08	3.97	8.91	52.41	75.9	5.16
IV	6.3	37.2	218.5	262	27.04	6.2	17.85	91.1	115.1	12.44
V	6.1	36.0	209	251.1	23.95	5.8	14.76	89.8	110.4	11.26
VI	5.93	32.7	204.5	243.2	30.57	5.7	14.08	82.3	102.1	10.09