Design of an Automation Framework for a Novel Data-Flow Processor Architecture

Karthick Lakshmanan

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

July 21, 2010
Blacksburg, Virginia

Keywords: Electronic Textiles, Automation Framework, Data-Flow
Copyright 2010, Karthick Lakshmanan

Design of an Automation Framework for a Novel Data-Flow Processor Architecture

Karthick Lakshmanan

ABSTRACT

Improved process technology has resulted in the integration of computing elements into multiple application areas. General purpose micro-controllers are designed to assist in this integration through a flexible design. The application areas, however, are so diverse in nature that the general purpose micro-controllers may not provide a suitable abstraction for all classes of applications. There is a need for specially designed architectures in application areas where the general purpose micro-controllers suffer from inefficiencies. This thesis focuses on the design of a processor architecture that provides a suitable design abstraction for a class of periodic, event-driven embedded applications such as sensor-monitoring systems. The design principles of the processor architecture are focused on the target application requirements, which are identified as event-driven nature with concurrent task execution and deterministic timing behavior. Additionally, to reduce the design complexity of applications on this novel architecture, an automation framework has been implemented. This thesis presents the design of the processor architecture and the automation framework explaining the suitability of the designed architecture for the target applications. The energy use of the novel architecture is compared with that of PIC12F675 micro-controller to demonstrate the energy-efficiency of the designed architecture.

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

I dedicate this thesis to my parents Dr.Lakshmanan Chockalingam and Mrs.Vijaya Lakshmanan.

Acknowledgements

The work presented in this thesis could not have been accomplished without the help of many people. I would like to thank each one of them who contributed towards the success of this research.

I would like to thank Dr. Mark Jones for being the advisor for my thesis research. His insightful ideas helped me a lot in difficult situations during the course of this research. I would also like to thank Dr. Thomas Martin for his extended support throughout this research. His *Wearable and Ubiquitous Computing* course got me interested in e-textiles research and helped me to get involved in this interesting research area.

I would like to express my gratitude to Dr. Paul Plassmann for serving in my thesis committee and for his valuable comments on my thesis presentation.

Special thanks to my friend and project mate Ms. Ramya Narayanaswamy for her continuous support throughout this research, without which this thesis would not have been possible. I would also like to thank all my friends who helped me by patiently reviewing this thesis.

I would not be here, but for the motivation and support provided by my brother Mr. Balasundararaman Lakshmanan. I am grateful for his thoughtful advice that encouraged me to pursue graduate studies.

Finally, I would like to thank all of my family members for their unconditional love and encouragement that motivated me towards the successful completion of my Masters degree.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Contributions 3

1.3 Thesis Organization 4

2 Background **5**

2.1 E-Textiles Applications 5

2.1.1 Wearable Motherboard 6

2.1.2 Acoustic Beam-Former 6

2.1.3 Activity Recognition 7

2.2 Related Research 7

2.2.1 System-Level Architectural Abstractions 7

2.2.2 Software-Level Architectural Abstractions 10

2.2.3 Processor-Level Architectural Abstractions 11

2.3 Introduction to Model-Driven Engineering 13

3 **Processor Architecture Design** **15**

3.1 General Processor Architecture 16

3.1.1 Network Interface 17

3.1.2 Coarse-Grained Modules 18

3.1.3 Interconnection Bus Network 18

3.1.4 Input FIFO and Output FIFO 19

3.2 Module Design 20

3.2.1 Input Register(s) 21

3.2.1.1 Packet Structure 22

3.2.1.2 Input Register Design 24

3.2.2 Wrapper Configuration Registers 25

3.2.2.1 Mod_active 26

3.2.2.2 Output Register Delay Array 27

3.2.2.3 Output Register Destination Array 28

3.2.2.4 Num_used 28

3.2.3 Internal Module 28

3.2.3.1 Internal Configuration Registers 29

3.2.4 Output Register(s) 30

3.3 CAM-based Generic State-Machine Design 31

3.4 Sample Timing Diagram 32

4 **Automation Framework Design** **34**

4.1 Automation Framework Outline 35

4.2 Architecture and Application Representation 37

4.2.1 Architecture Representation 37

4.2.1.1 Architecture Description Language 38

4.2.2 Application Representation 39

4.3 Prototyping Framework 39

4.3.1 ADL Compilation 40

4.3.2 Automatic Design Generation 41

4.3.3 Tool-Flow Automation 41

4.4 Configuration Framework 41

4.4.1 Mapping 42

4.4.1.1 Mapping Algorithm 42

4.4.2 Scheduling 42

4.4.2.1 Scheduling Algorithm 44

4.5 Validation Framework 44

4.5.1 Automatic Test-Bench Generation 45

4.5.2 Simulation and Results Extraction 45

5 Validation and Results **46**

5.1 Experimental Setup 46

5.2 Experimental Applications and Architectures 47

5.2.1 Applications 47

5.2.2 Architectures 47

5.3 Schedule Validation 48

5.3.1 Validity of the Generated Schedule 48

5.3.2 Resource Sharing, Pipelining, and Concurrency 51

5.3.3 Configurability Analysis 52

5.4 Energy Usage Analysis 53

5.4.1 Energy Usage vs. Bus Width 53

5.4.2 Energy Usage of Template 54

5.4.3 Energy Usage vs. Resource Sharing 58

5.4.4 Energy Usage Comparison with PIC Family 60

6 Conclusions and Future Work **63**

6.1 Future Work 64

Bibliography **66**

A Application Graphs **70**

B Test Architectures **72**

C UML Description of Application and Architecture Objects **74**

C.1 Architecture Class 74

C.2 Application Class 76

D Sample Application Description **80**

E PIC Implementations **82**

List of Figures

2.1 Two-tier System Architecture 9

3.1 Overall Processor Architecture 17

3.2 Template Structure of a Functional Module 21

3.3 Twelve-bit Packet Structure 22

3.4 Functional Bits in a Data Packet 22

3.5 Functional Bits in a Wrapper Configuration Packet 22

3.6 Functional Bits in a Internal Configuration Packet 23

3.7 Input Register Top Level Design 25

3.8 Wrapper Configuration Block Top Level Design 26

3.9 Internal Module Top Level Design 29

3.10 Output Register Top Level Design 30

3.11 Index Formation in SME 31

3.12 Sample Timing Diagram 32

3.13 Application Graph for Thermostat Application 32

4.1 Framework Overview 35

4.2 Architecture Model for Architecture-I 37

4.3 Application Graph for Application-I (Thermostat) 40

5.1 Schedule Diagram For Multiplier Application On Architecture-III (Part-1) . 49

5.2 Schedule Diagram For Multiplier Application On Architecture-III (Part-2) . 50

A.1 Application Graph for Application-II (Multiplier) 70

A.2 Application Graph for Application-III (Hypothetical) 71

B.1 Architecture Model for Architecture-II 72

B.2 Architecture Model for Architecture-III 73

C.1 Architecture Class UML Representation 75

C.2 Application Class UML Representation 77

List of Tables

5.1 Configuration Values for Application-II on Architecure-III 52

5.2 Configuration Values for Application-III on Architecure-III 52

5.3 Schedule Length vs. Bus Width 54

5.4 Bus Width vs. Area 54

5.5 Bus Width vs. Energy Usage 55

5.6 Area and Energy Cost of Template Components 55

5.7 Area and Energy Cost of Three Different Templates 56

5.8 Template vs. Internal Module 57

5.9 Template-I Area vs. Bus Width 57

5.10 Template-I Energy Cost vs. Bus Width 58

5.11 Resource Sharing vs. Energy Usage 59

5.12 Resource Sharing vs. Wrapper Configuration 60

5.13 Energy Usage Comparison for Application-I 62

5.14 Energy Usage Comparison for Application-II 62

Chapter 1

Introduction

Over the past few decades computing has become integrated into the lifestyle of many people. Computing technologies have even reached the stage of being woven into the fabrics that people wear, making true the words of Mark Weiser, "The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it." [1] Throughout this evolution, from huge mainframe computers to very small integrated elements within fabric, computing elements have increased in complexity. Furthermore, the constraints in energy and area have become more stringent.

General purpose processors, such as PIC micro-controllers [2], have improved in speed and capability to support this evolution. Nevertheless, they still fall short for certain applications, particularly for applications in which energy constraints are critical. These applications are inherently event-driven in nature. In contrast, the target architectures are control-flow sequential processors. This results in an incompatibility between the application domain and target architecture. In such a situation, general purpose processors sacrifice efficiency for generality. Thus, there is a need for a specially designed power-aware processor, with decreased design complexity, for such applications.

This thesis discusses the design of such a processor architecture. Event-driven embedded

applications such as sensor-monitoring map well onto the novel data-flow based processor, designed without an Instruction Set Architecture (ISA), resulting in energy efficient implementations coupled with reduced design complexity.

1.1 Motivation

Electronic Textiles (E-Textiles) refer to textiles that have integrated computing elements, such as sensing, processing, and actuating nodes, woven into them. A typical application of an e-textile collects data from multiple sensing nodes at specified intervals and performs tasks based on the collected data. Often, the primary aim of the sensing nodes in a sensor network is to collect data from sensors, perform simple manipulations on them and send the processed data to a node with higher computing capability. The sensing node application is inherently event-driven involving concurrent operation of multiple tasks. Consequently, mapping these applications on the traditional ISA based sequential processor is complex. Thus, the use of a general purpose processor for the implementation of these sensing nodes results in an inefficient implementation in terms of energy usage and performance. Moreover, due to the incompatibility between the application structure and the architecture, the complexity of implementation increases. This results in increased time for modeling, implementation, and validation of applications.

To bridge the gap between the application and the architecture, a software framework is typically implemented over the architecture. This software framework will act as an interface for mapping the applications onto the architecture [3]. However, these software frameworks have associated overheads which result in performance degradation with increased energy usage per computation. Nevertheless, these inefficiencies are not affordable in an energy-constrained environment such as e-textiles. Another approach to bridge the gap between application and architecture is to design a processor architecture that will suit the event-driven nature of the applications [4]. With the traditional ISA approach for an event-driven

processor architecture, however, the programming complexity increases.

Thus, this work aims to identify a suitable processor architecture for a set of applications without the traditional ISA approach. Additionally, this work seeks to develop an automation framework for the identified target architecture. In particular, the design of the automation framework aims to speed up the modeling of applications facilitating rapid prototyping and validation of target architectures.

1.2 Contributions

This thesis describes three primary contributions towards the design of an energy-efficient processor with less design complexity for a specific set of event-driven applications.

The first contribution is the design and implementation of a novel delay-based non-ISA processor architecture for event-driven distributed embedded applications. The architecture design involves a data-flow approach with coarse-grained hardware modules. The design of the hardware modules in the architecture is made reusable by having a template wrapper structure for all the modules with a core functional part. Moreover, each module is designed with a three-stage pipeline for energy-efficient use of resources. Handshake signals are implemented between these stages to allow for a power-aware design.

A further contribution is the design and implementation of an object-oriented automation framework for architecture and application modeling. An Architecture Description Language (ADL) is defined to easily specify the target architecture design. From the higher level ADL description of the architecture, the hardware design files are automatically generated leading to rapid prototyping of the target architectures. The application is represented as a Directed Acyclic Graph (DAG) depicting the data-flow between the coarse-grain nodes of the application. Additionally, for a given test stimulus, the generation of test bench to validate the implementation is automated.

The third contribution is the automation of mapping an application graph to the specified architecture design and scheduling the application on the architecture. The scheduling algorithm is implemented to enable resource sharing between multiple application nodes. Also, the algorithm design makes use of the three-stage pipelining in the hardware resulting in a schedule with higher bus usage.

1.3 Thesis Organization

The thesis is organized as follows. Chapter 2 discusses the background information necessary for understanding the work performed in this research. Chapter 3 discusses the details of the novel data-flow non-ISA processor architecture. Chapter 4 explains the design of the automation framework with details of the components of the framework. Chapter 5 discusses the results obtained through sample designs and analyzes the effectiveness of the architecture for the specified applications in terms of energy efficiency. Chapter 6 summarizes the contributions drawing out certain conclusions based on experimental results and identifies future areas of work taking this research forward.

Chapter 2

Background

This chapter discusses the background information related to this research. Several sensor-monitoring based e-textiles applications are introduced, illustrating the factors that necessitate a specialized processor architecture. This is followed by a brief description of research progress towards achieving energy efficiency and easier design capability for event-driven embedded applications. Additionally, a brief introduction to the basic concepts behind the data-flow design of the processor is given. Finally, the work done in this thesis is briefly described, differentiating the work from related research.

2.1 E-Textiles Applications

Textiles have long been an integral part of human life. Therefore, it is a natural extension to embed computing elements within textiles. This embedding helps to leverage the power of computing elements with the flexibility of the customary platform of textiles. This flexibility of textiles has led to research that aims to integrate electronics and textiles to achieve a platform called e-textiles [5, 6, 7]. Due to the unobtrusive nature of e-textiles, researchers believe that e-textiles are a suitable platform for pervasive computing [8].

In this section, three example e-textiles applications are discussed. In all the applications, the basic design has a common skeleton structure. All the applications have a set of sensing nodes, with attached sensors, which are connected through a wired-network embedded within the fabric. With a wired network, the energy cost associated with data communication is reduced. This implies that the processing elements are the main power consuming components of the system.

2.1.1 Wearable Motherboard

The "Wearable Motherboard" project at Georgia Tech has been developed as an integrated system to monitor vital signs from the wearer's body, acting as a flexible framework for "personal mobile information processing"(PMIP) [9]. The "Wearable Motherboard" consists of a set of sensor nodes and interconnect structures integrated within the fabric creating a "wearable integrated information infrastructure". This network of nodes gathers information from various parts of the body and communicates it to the appropriate systems for processing.

2.1.2 Acoustic Beam-Former

Past e-textiles research at Virginia Tech has covered a range of aspects aiming to identify the key challenges in e-textiles development. For example, e-textiles lab at Virginia Tech has developed a design framework for e-textiles [10]. Two applications [11] have been simulated using the design framework. These designs are then validated by developing prototypes. The first prototype is a large scale fabric with a collection of microphone arrays. The application aims to identify the location of a distant moving vehicle with the help of the sound waves received by the microphone arrays. A collection of microphones connected to a micro-controller is called a cluster. The fabric consists of multiple clusters interconnected by a network embedded in the fabric.

The second prototype is a speech processing vest. The aim of this application is to identify

the direction of the speaker in a teleconferencing room. This application also has a similar design with a set of microphones embedded on a fabric. In this design, the relative time difference in the reception of sound signals is used to identify the direction of the speaker.

2.1.3 Activity Recognition

In [12], a self-contained motion sensing e-textile garment is described. This garment acts as an activity recognition system that can be used for a variety of applications, such as health monitoring, military applications, and entertainment industry. The system consists of a collection of Tier-1 nodes co-located with sensors to collect motion-sensing data. The data collected by the Tier-1 nodes is transferred to a single Tier-2 node which analyzes the data to recognize the activity of the wearer. The hierarchical architecture of nodes is explained in detail in the next section.

2.2 Related Research

As explained in the previous section, in e-textiles applications the power consumption of the processing elements is a significant component of the overall power consumption of the system. Thus, it is important to reduce the power consumed by the sensing node processing elements. The following sections discuss about various research efforts towards this end at different levels of abstractions.

2.2.1 System-Level Architectural Abstractions

The design decisions made in the overall system architecture impact the design complexity and performance of individual components of the system. To this end, the design framework developed at Virginia Tech [10] modeled various aspects of the system, such as the number and location of sensors, the number and location of processors, and the communication

network topology. Through this framework, different models of the system can be simulated to evaluate critical metrics such as power consumption and sensing accuracy. Thus, even without designing expensive prototypes, design decisions can be made to improve system behavior.

The overall system architecture plays an important role in the efficient functioning of any system. Towards this end, specialized system architectures are designed for specific applications. One such specialized system architecture has been proposed for wireless sensor network based structural monitoring application [13]. Traditionally, the structural monitoring systems have been a star wireless network of distributed sensor units communicating to a centralized monitoring station. Kottapalli *et al.*, in [13], suggest a hierarchical two-tier system architecture for structural monitoring. As per the suggested system architecture, each sensor location is a cluster of sensor units with a local site master. These local site masters communicate the data to the central site master. As the distance of wireless communication is reduced, the power consumption for data-communication is reduced. This design uses a smaller number of low-tier processing nodes *via* time-based multiplexing of the sensor units to reduce cost.

The same approach can be translated to e-textile applications, because of the similarities between the nature of these applications. In [14], a similar architecture is proposed for e-textile applications. Jones *et al.*, in [14], present the advantages of having a digital wired network embedded within the fabric, compared to a wireless sensor network. Moreover, a two-tier heterogeneous system architecture is suggested for e-textile applications. Figure 2.1 shows a representation of the suggested architecture. This architecture has two tiers of processing nodes. The Tier-1 nodes are the sensing nodes that gather data from the co-located sensors and perform simple operations on the data. The Tier-1 nodes then communicate the processed data in digital form to the on-fabric wired network. The Tier-2 nodes are high performance processors that manage the network. These nodes collect the data from Tier-1 nodes and execute the actual application. This type of system architecture is shown to have better reliability and scalability. Additionally, with decentralized sensor control, a power-

aware design is easier to implement. In [15], a prototype motion sensing e-textile jumpsuit is designed as a proof of concept for the suggested two-tier system architecture.

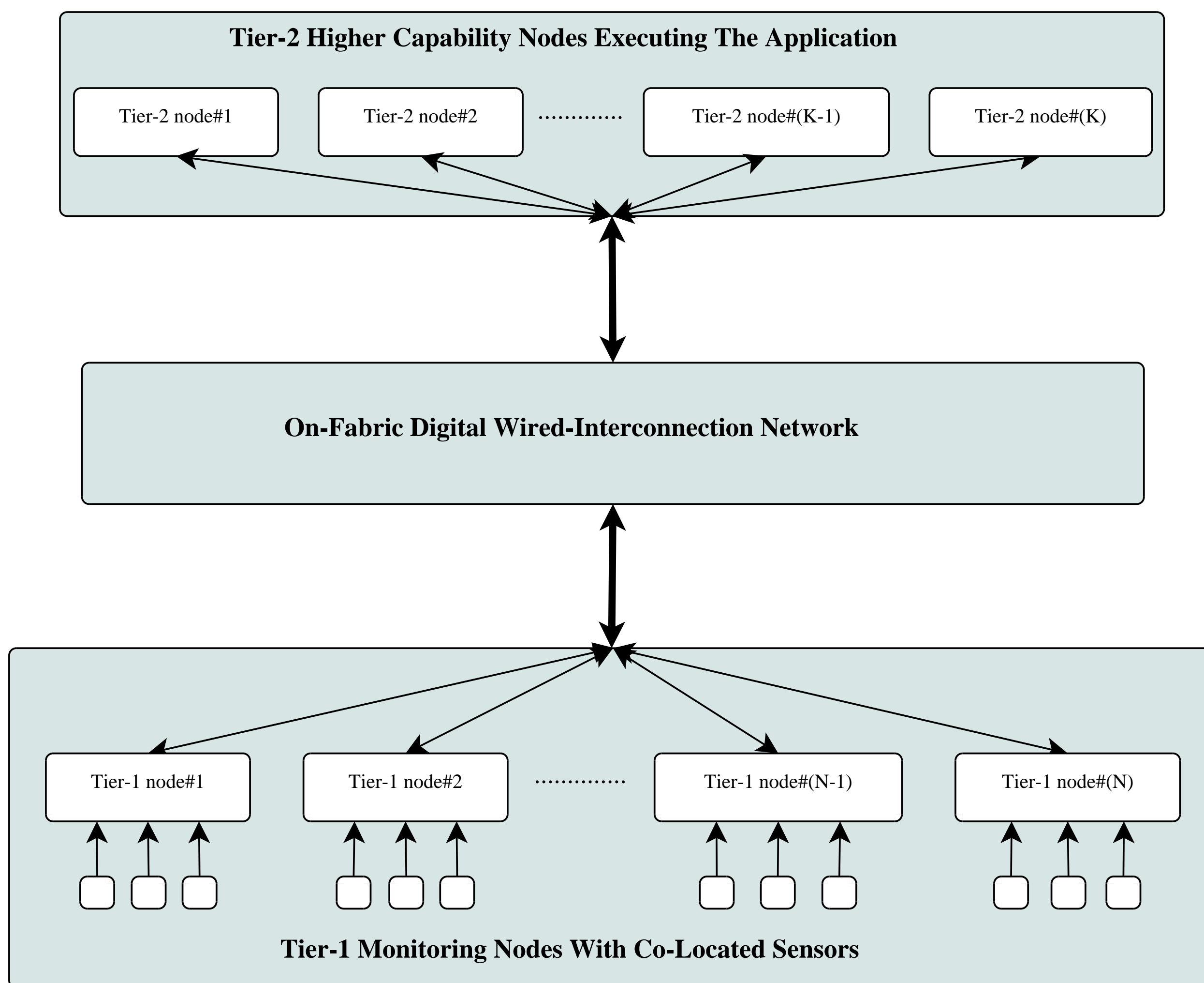

Figure 2.1: Two-tier System Architecture

Although these architectural modifications result in better implementations at a system level, there are still issues in the efficient implementation of Tier-1 sensor node applications. This is because of the mismatch between the general purpose processor architecture and the sensing applications. Therefore, a target processor architecture specifically designed for Tier-1 sensing node applications is needed [14]. Thus, as an extension of the two-tier system architecture research, this thesis aims to design a data-flow processor architecture specifically for Tier-1 sensing node applications.

2.2.2 Software-Level Architectural Abstractions

To reduce the design complexity of applications due to the mismatch between application and target architecture, typically a software abstraction is provided for the target hardware. This software abstraction will act as a mapping interface between the application and the target architecture to bridge the gap between the two. For example, there are commercial operating systems specifically tailored for real-time embedded systems [16, 17]. These operating systems use a hardware abstraction layer that acts as an interface over which platform independent applications are developed. The hardware abstraction layer is customized specifically for each target architecture and aims to take full advantage of the underlying hardware. However, these commercial operating systems target general purpose processors.

E-Textile based sensor-monitoring applications demand a specialized software abstraction that suits the event-driven nature of the applications. TinyOS is an open source initiative that provides a software abstraction designed specifically for event-driven embedded applications to be mapped over a representative class of general purpose micro-controllers [3]. Support for specially designed processor architectures are extended through updated versions [18].

Sensor node applications are typically event-driven with multiple tasks running concurrently. Taking into account the nature of target applications and architectures, TinyOS reduces event-propagation and context-switching complexity. The entire OS fits in a small-sized memory. TinyOS is implemented in a modular way with coarse-grained components and a task scheduler. The task scheduler is a simple Bounded-FIFO scheduler with preemption. Complex schedulers are implemented specifically based on the needs of the application. Tasks are scheduled based on the occurrence of events. Though the software abstraction reduces the design complexity, it does not give a fine-grained control over the target architecture. This results in an energy inefficient design with performance degradation due to the overheads associated with context-switch and event-scheduling [19].

In [20], a hybrid three layered hardware abstraction is suggested. The three layers of abstrac-

tion are Hardware Presentation Layer (HPL), Hardware Adaptation Layer (HAL), and Hardware Interface Layer (HIL). HPL is the direct interface with the hardware, which presents the capabilities of the hardware in native concepts of operating systems. HAL represents the actual architecture abstraction consisting of coarse-grained components. This provides a more useful higher level abstraction hiding the complexity of HPL primitives. HIL is the topmost abstraction that translates the platform specific abstraction provided by HAL into hardware independent interfaces through platform independent abstractions. HAL interface is used in components where performance and energy-efficiency is important. The rest of the components are interfaced through the platform independent HIL abstraction sacrificing performance and energy efficiency.

Though this approach improves the portability of the implementations, the abstraction should match the hardware intricacies of the platform for efficient implementations [21]. Thus, there is a need for a target architecture that inherently supports the event-driven nature of applications providing finer control over event-handling. The designed processor architecture seeks to achieve the same.

2.2.3 Processor-Level Architectural Abstractions

Designing a specialized processor architecture is the lowest-level of design enhancement to achieve energy-efficient and less complex implementations. Prior research such as [4] and [19] seek to design a custom processor architecture for wireless sensor networks.

In [4], the design of an event-driven processor named Sensor Node Asynchronous Processor (SNAP) is explained. As the name suggests, the processor design is asynchronous and clock-free. Handshake signals are implemented to handle synchronization. To adapt to the event-driven nature of the applications, the processor has a hardware event queue and event co-processors. Events can be either internally generated through a timer co-processor or externally generated network messages which are managed by message co-processors. The hardware event queue and co-processors avoid the necessity for a software event-scheduling

operating system resulting in less performance overhead. Additionally, the asynchronous design of the processor leads to an automatic fine-grained power management as there is no switching activity in the parts that are not active. SNAP uses an ISA as described in [22]. This ISA has special extensions for each of the co-processors and for event-handling. SNAP does not have virtual memory or interrupts. SNAP has a predictable timing behavior as events are scheduled through event-queues without any preemption.

Despite having native support for event-scheduling in SNAP, the primary computing engine and the co-processors are still inherently sequential due to the ISA design. The asynchronous design of the SNAP architecture needs an additional circuitry for synchronization, thus consuming more power. Moreover, the functional units in SNAP have little pipelining resulting in reduced throughput. In contrast, this thesis aims to design a non-ISA based processor architecture that inherently supports concurrent execution of tasks. With a non-ISA design, the processor is programmed using simple configurations that are automatically generated by the designed automation framework. Additionally, the functional modules of the architecture designed in this thesis are internally pipelined, thus increasing the throughput.

In [19], a coarse-grained event-driven architecture is designed. This architecture is designed in a modular way resulting in a scalable implementation. The coarse-grained modules are optimized for frequently used tasks, providing hardware acceleration. There are multiple coarse-grained modules, such as Analog-to-Digital Converters (ADC), filters, timer unit (that generates periodic events), message processor (that handles message transfers), and radio component (that handles wireless communication). These modules act as slave-components that generate events. This architecture consists of a general purpose micro-controller and an event-processor that act as master-components, which schedule the events generated by the slave-components. Fine grained power-management is provided through explicit programmer commands, which should be implemented in the Interrupt Service Routine(ISR) based on application requirements.

Though this architecture is well suited to the event-driven paradigm, the general purpose

controller and the event-processor used for event-scheduling consume power throughout the application execution. Moreover, a power-aware design requires programmers' effort. In contrast, the processor architecture designed in this thesis controls the event-scheduling through configurations of the coarse-grained modules, thus avoiding the necessity for an event-processor. Additionally, the handshake signals in the design can be used by a hardware power manager that automates the power management.

The event-handling overhead can be avoided using a data-flow approach. The Elemental Computing Architecture(ECA) is one such computing paradigm which is based on the data-flow approach [23]. ECA is a reconfigurable architecture designed for highly parallelizable tasks such as Software Defined Radios(SDR). The architecture is designed with fine-grained elements similar to an FPGA architecture. The elements are heterogeneous in nature and can be configured to perform different operations based on different contexts. The elements wait on an input queue and execute once all the inputs are available. Similarly, there is an output queue at the other end and execution also depends on the availability of space in the output queue. Thus, the execution of elements is totally controlled by the hardware, based on the state of input and output queues. This avoids the need for an event-handling scheduler in software or a separate event-processor in hardware. Nevertheless, the communication between elements is non-deterministic depending on the availability of space in the input and output queues. In contrast, the data-flow architecture designed in this thesis supports deterministic static scheduling of events through configurations.

2.3 Introduction to Model-Driven Engineering

Apart from the energy-efficiency constraints, the other aspect that this thesis concentrates upon is decreased design complexity for the implementation of applications over the designed architecture. This thesis aims to reduce the design complexity through Model-Driven Engineering(MDE). For complex embedded systems, MDE helps to reduce the design com-

plexity and design time by providing a higher design abstraction at the application level. Moreover, MDE helps in validation of the design at a early stage, reducing the design iterations, design cost and design time. There are already existing commercially successful MDE products that are used in safety-critical industries such as Automotive and Aeronautical Engineering [24, 25].

Simulink is a comprehensive environment for model-based design and simulation of complex embedded systems [24]. Reactis provides model-based tools to automatically test and validate the model generated for the application [25]. These commercial tools have the capability to automatically generate the application program and test cases for various target architectures. The main issue with this approach is that the automatically generated code is inefficient in terms of performance and energy usage. This necessitates manual editing of the auto generated code for performance improvement and energy efficiency which leads to one full software development cycle. This is mainly because of the mismatch between the target architecture and the computational model used in MDE. The modeling environments use a data-flow approach to model the behavior of applications. Therefore, compared to the control-flow based ISA processor architectures, the designed processor architecture is a better target for MDE.

Chapter 3

Processor Architecture Design

This chapter describes the design of the novel delay-based data-flow processor architecture. As emphasized in Chapter 1, the architecture is designed without an ISA. The architecture is designed with native support for data-flow based asynchronous event-handling, pipelining, and flexible resource-sharing. Handshake signals are implemented to manage the asynchronous event-flow within a module. These handshake signals can be tapped by a power manager to yield a power-aware design. Additionally, the architecture is designed to require limited configuration, simplifying the programming of the processor compared to the traditional ISA based processors.

To begin with, a brief explanation about the overall architecture is given. This is followed by a detailed description of the design of individual components of the architecture. Finally, the design of a generic State-Machine Engine (SME) module based on Content Addressable Memory (CAM) is discussed.

3.1 General Processor Architecture

This section describes the overall processor architecture design at a higher level of abstraction. The main components of the designed processor architecture are functional modules, event-bus network, and FIFOs. The functional modules are coarse-grained configurable modules that perform a defined function. The functionality of the coarse-grained module is calibrated to the application specific behavior through configuration-events. These functional modules are interconnected through an event-bus network that transfers the data-events between the functional modules. The functional module starts functioning only after it receives all the inputs and generates output data-events. The output data-events are transferred to their destination on predefined cycles through the event-bus network. The FIFOs act as event-queues, managing the transfer of events between different clock-domains. Additionally, the FIFOs act as the configuration channel, transferring the configuration events from the "outside world" to the functional units.

The current version of the designed architecture supports a single event-bus, with two event-FIFOs (input FIFO and output FIFO), and a collection of functional modules. The architecture design is parameterized with control over the bus width. Similarly, the design of the functional modules is parameterized for flexible design of architecture instances. Different instances of architectures can be generated by varying the number of functional modules in the architecture and the choice of the functional modules.

Figure 3.1 shows the design of an example instance of the architecture. The designed processor architecture consists of a network interface, an input FIFO and an output FIFO to manage the communication between the architecture and network interface, and a set of coarse-grained functional modules. The coarse-grained functional modules are connected through an interconnection bus network. Each module has a hardware address associated with it. As both the input and output FIFOs are unidirectional, they share a common hardware address to be used for communication in each direction.

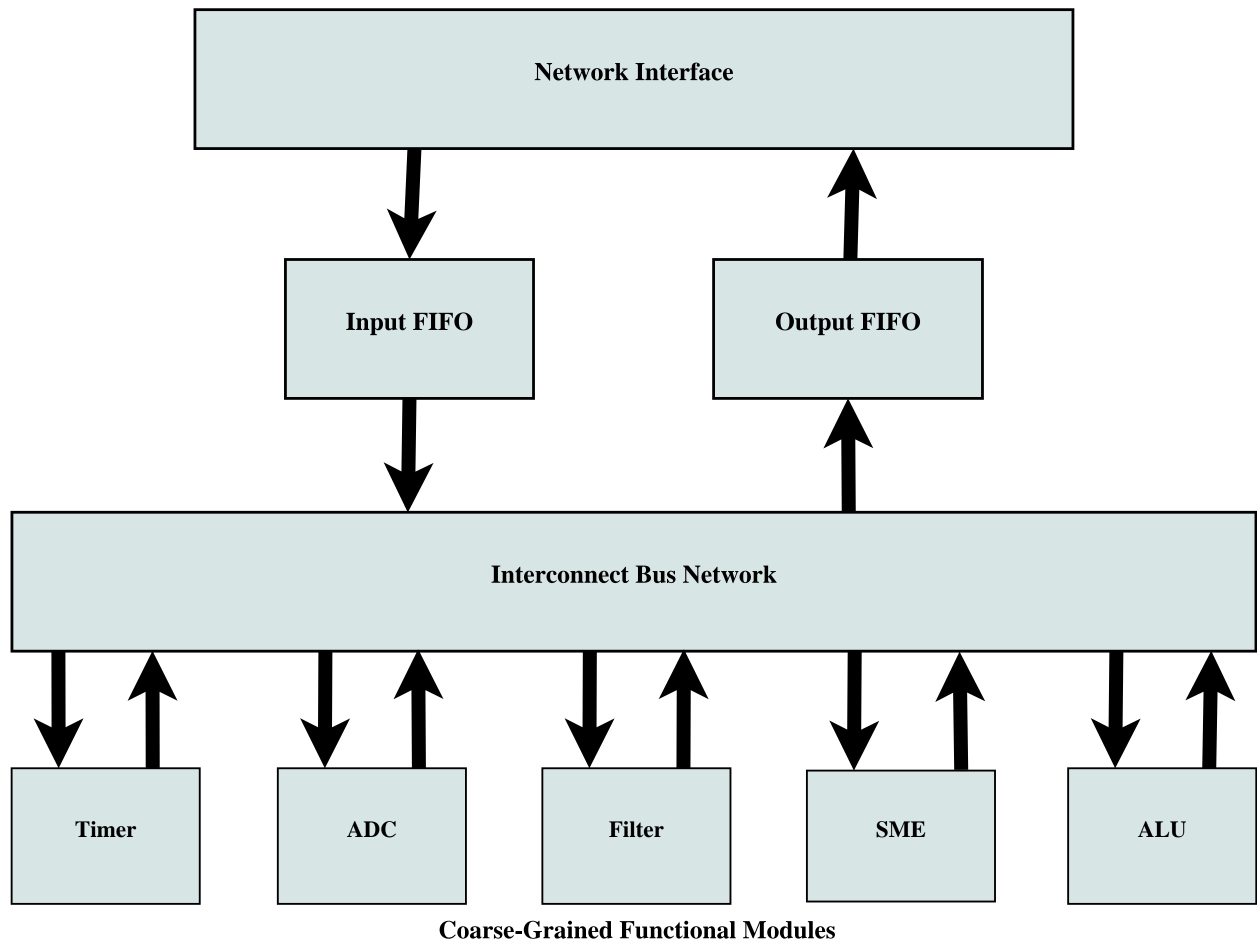

Figure 3.1: Overall Processor Architecture

3.1.1 Network Interface

The network interface acts as a point of communication between the processor architecture and the "outside world" to which the processor is connected. The communication interface does the packet translation between the processor and the network in either direction of communication. Therefore, the design of the interface will vary depending on the network protocol. Because, the main focus of this thesis lies in the actual design of the processor, the design of the interface is not considered here.

3.1.2 Coarse-Grained Modules

The coarse-grained modules form the functional part of the processor architecture. These modules are hardware implementations of the functions used in target applications. Typical functional modules that are used in sensor-monitoring applications include a timer unit to generate timed periodic events, an ADC for sensor-reading, a filter for data manipulation, an arithmetic unit to perform simple operations on data, and an SME to implement control structures.

The coarse-grained modules are configured and operate independently. The design of the coarse-grained module can be optimized for target applications. The advantage of this approach is that the optimization for energy and performance can be implemented at the hardware level itself, offloading the effort from the programmer. Moreover, the implementation of a power-aware design is simpler with isolated, functionality-based modules.

3.1.3 Interconnection Bus Network

The coarse-grained modules are connected through an interconnect bus network. The time-sharing of this bus network is deterministic, controlled through delay configurations which is explained in Section 3.2.2. As the time-sharing of this bus network is deterministic, there is no need to implement arbitration logic in the bus network, thus, leading to a simpler energy-efficient design. This deterministic design allows for guarantees and predictability for real-time bounds, correct operation and power. This bus network acts as an interface for communication between modules, transferring the data packets that act as data-flow events. Additionally, the bus network acts as an interface between a module and the network interface in both directions, transferring the configuration and asynchronous events from the network interface to the modules, and transferring the data from the modules to the network interface.

The bus network is the part of the architecture that is active most of the the time. Therefore,

it is important to have a design for the bus network that targets energy-efficiency. To this end, the bus network is designed in a way that it does not differentiate between configuration and data events. This differentiation is managed by the individual modules. All of the coarse-grained modules have a template structure as described in Section 3.2. Thus, the design of the bus network is oblivious to the underlying complexity of the coarse-grained modules, yielding a uniform interface to all of the modules. The design of the overall architecture has bus width as a parameter, which is an important design decision for the architecture.

3.1.4 Input FIFO and Output FIFO

The input and output FIFOs, in the current architecture instance, are two unidirectional event-FIFOs that manage the communication between the network interface and the functional modules. The input and output of the FIFOs operate in different clock-domains acting as a synchronizing medium between the two clock-domains. Often, the network operates at a higher clock frequency than the functional modules. Given the target application requirements, the functional modules are intentionally designed to operate at a lower clock frequency.

The input FIFO communicates the data from the network interface to the functional modules through the bus network. This FIFO acts as a communication interface for configuration events from the network. Additionally, the input FIFO communicates the asynchronous events from the network to the appropriate functional module. Because the interconnect bus network is time-shared between the functional modules, the Input FIFO transfers the data to the bus network only in the specified cycles of operation. This behavior is controlled by three specific configurations for the input FIFO, namely, *busy_cycles*, *free_cycles*, and *pattern_mask*.

Busy_cycles configuration indicates the number of busy cycles in a single schedule period of the application during which the input FIFO does not transfer data to the bus network. *Free_cycles* configuration indicates the number of free cycles after the schedule period during

which the input FIFO transfers re-configuration and asynchronous events to the bus network. The input FIFO has an internal counter to keep track of the busy cycles of operation. The initialization of this counter is handled by the *pattern_mask* configuration. The *pattern_mask* configuration has the first $four$ bits that will be transferred onto the bus network at the start of the schedule. The input FIFO keeps track of the first $four$ bits of each packet transferred onto the bus network. If the bits match the *pattern_mask* configuration value, then, the counter is initialized. With this approach, the input FIFO does not disrupt the application schedule.

The output FIFO has a simple design that immediately transfers the data from the bus network to the network interface. The FIFO design is parameterized with the "depth of the FIFO" as a parameter.

3.2 Module Design

This section describes the design of the individual coarse-grained modules. Though each individual module can vary in functionality, the design of these modules has a fixed template structure as given in Figure 3.2. This fixed template structure results in a uniform bus interface for each of the functional module. The functional module has *MOD_ID* as a parameter that indicates the hardware address of the module. These parameters are implemented for design flexibility and are fixed for a particular architecture instance. These are not run-time configurations which can be changed based on application requirements.

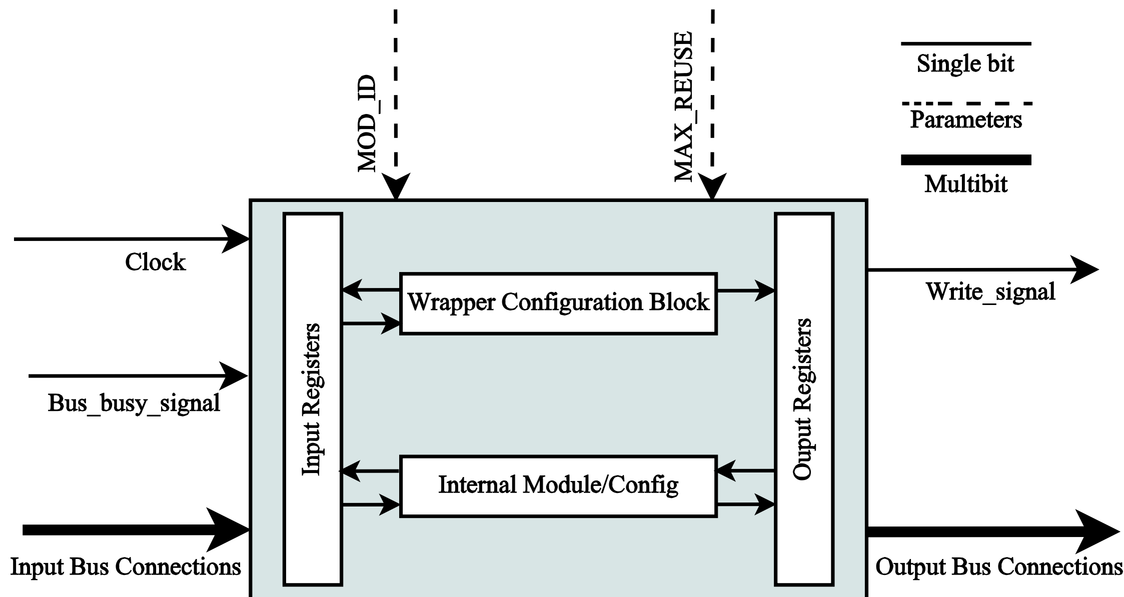

Figure 3.2: Template Structure of a Functional Module

The subsequent sections explain the design of the components of a functional module. The key components within a functional module are input registers, output registers, wrapper configuration registers, internal configuration registers, and the internal module. The internal module implements the actual functionality of the coarse-grained module such as addition, subtraction, and A/D conversion. These components are designed as a template and, except for the internal module, components can be used by any of the functional modules by simply instancing these templates.

3.2.1 Input Register(s)

Each of the functional modules will have one or more input registers connected to a bus. Even modules such as timers, which do not receive any data input, have at least one input register to receive the configuration events.

The main functionality of the input register is to read the bus, identify whether a packet is addressed to the particular module, and transfer the packet to the next component inside the module. The address of the module is parameterized using the *MOD_ID* parameter. If

there is more than one input register, the registers are grouped together as an input register file and will transfer the packets to the next level only when all the necessary input values are available.

A module receives three different types of packets, namely, wrapper configuration, internal configuration, and data packets. The input register identifies the nature of an input packet through the specific bit values that indicate whether it is a wrapper configuration, internal configuration or a data packet. Thus, to understand the design of the input register, it is necessary to know the packet structure of the designed architecture.

3.2.1.1 Packet Structure

This section illustrates the packet structure followed in the current version of the architecture. The current design uses a *twelve*-bit packet structure as illustrated by Figures 3.3, 3.4, 3.5, and 3.6.

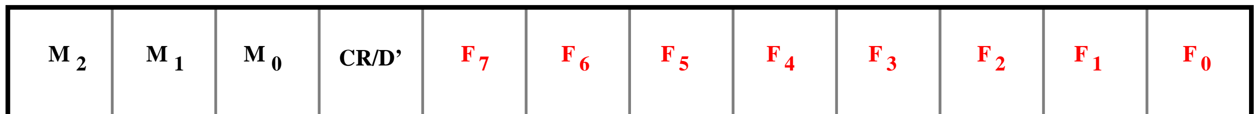

Figure 3.3: Twelve-bit Packet Structure

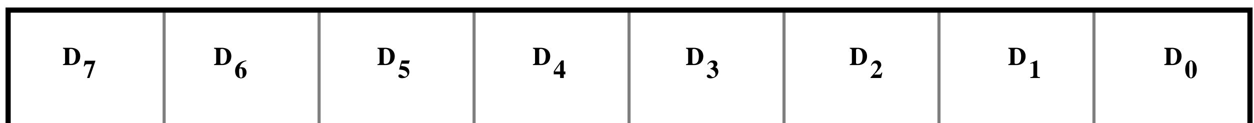

Figure 3.4: Functional Bits in a Data Packet

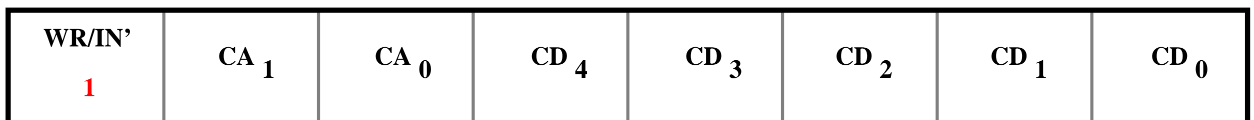

Figure 3.5: Functional Bits in a Wrapper Configuration Packet

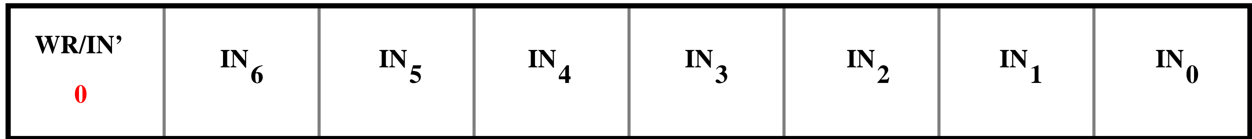

Figure 3.6: Functional Bits in a Internal Configuration Packet

The following description explains the functionality of each bit-notation given in the Figures 3.3, 3.4, 3.5, and 3.6.

$M_{2..0}$

These are module address bits that indicate the address of the destination device for the packet.

CR/D'

If this bit is *one*, it indicates that the packet is a configuration packet. If this bit is *zero*, it indicates that the packet is a data packet.

$F_{7..0}$

These are functional bits that vary in functionality based on whether the packet is a data, wrapper configuration or internal configuration packet.

$D_{7..0}$

These are *eight* functional bits in a data packet indicating the data value.

WR/IN'

This is the F_7 functional bit in a configuration packet. If this bit is *one*, it indicates that the configuration is for the wrapper. If this bit is *zero*, it indicates that the configuration is for the internal module.

$CA_{1..0}$

These are F_6 and F_5 functional bits of a wrapper configuration packet. These bits indicate the address of the wrapper configuration register to which the configuration data is addressed. A detailed explanation is available in Section 3.2.2.

$CD_{4..0}$

These are $F_{4..0}$ functional bits of a wrapper configuration packet. These bits give the value to be filled in the wrapper configuration register.

$IN_{6..0}$

These are $F_{6..0}$ functional bits of an internal configuration packet. These bits differ in functionality based on the internal module design and are specific to that particular functional module.

The packet structure is designed to achieve high bus utilization. Bits such as $M_{2..0}$, CR/D', and WR/IN' are overhead bits. The ratio of value bits to these overhead bits should be high. Data transfers occur far more frequently compared to configuration events. Thus, it is important to keep this ratio higher for data packets. The architecture achieves this through design customization that avoids having address bits to address the data to a particular input register in the module. Instead, the input registers are filled in a round-robin fashion avoiding the necessity for input register specific address bits.

3.2.1.2 Input Register Design

Figure 3.7 shows the top-level design of an input register showing the parameters, input signals, output signals, and handshake signals.

The input register reads the bus continuously looking for an input that is addressed to the module. The start of packet transmission is identified by the *bus_busy_signal*. The input register module has an internal counter that counts the number of times the register has to read the bus to form a packet. The number of times to read is calculated by dividing the packet-width (twelve) by the bus width. Once the input register receives the packet, it identifies the type of the packet and sets the *data_pkt_ready* or *int_cfg_pkt_rdy* or *wr_cfg_pkt_rdy* signal based on whether the packet is a data packet, internal configuration packet or wrapper configuration packet respectively. This signal is reset based on the reception of the corre-

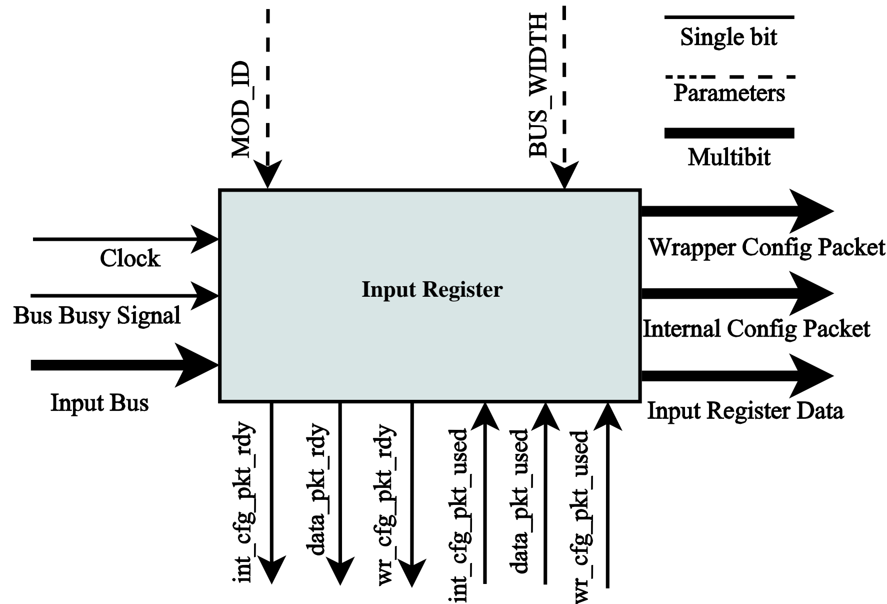

Figure 3.7: Input Register Top Level Design

sponding acknowledgment signal from the appropriate module (*data_pkt_used* from internal module, *int_cfg_pkt_used* from internal configuration module, or *wr_cfg_pkt_used* from the wrapper configuration block). These handshake signals manage the asynchronous data-flow between the components.

3.2.2 Wrapper Configuration Registers

The designed processor architecture is primarily programmed through configuration registers that control the behavior of the functional modules. There are two types of configuration registers, namely, wrapper configuration registers and internal configuration registers. wrapper configuration registers are those configuration registers that are common to all functional modules. Therefore, these configuration registers are added to the general template structure of the functional module. Figure 3.8 shows the overall structure of the wrapper configuration block.

During the configuration stage, the input register generates the *wr_cfg_pkt_rdy* signal along

with a wrapper configuration data. When the wrapper configuration block reads this signal, it copies the wrapper configuration data to the corresponding configuration register and generates a *wr_cfg_pkt_used* signal as an acknowledgment.

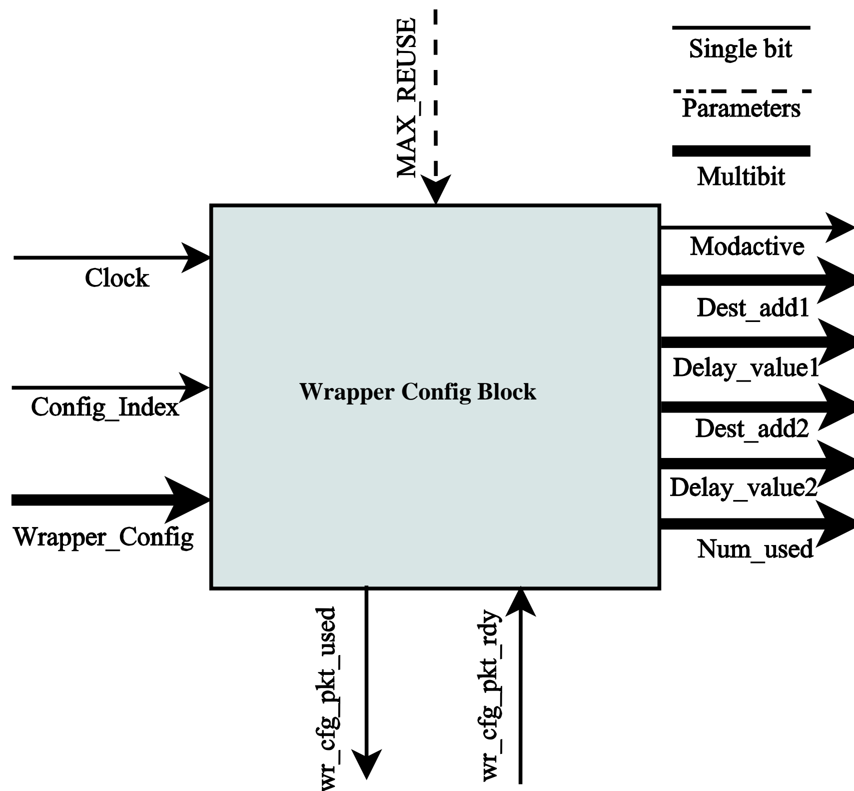

Figure 3.8: Wrapper Configuration Block Top Level Design

In the current design of the processor architecture, there are four wrapper configuration registers as explained in detail in the following sections.

3.2.2.1 Mod_active

This is a single bit wrapper configuration that indicates whether the particular functional module is active in the current application programmed on the architecture. The components of a functional module begin to operate immediately once the *Mod_active* configuration bit is set. Thus, when programming the architecture, the *Mod_active* configuration bit of the source functional module (usually a timer), should be configured last. *Mod_active* configuration is

also useful for a power-aware implementation, in which the inactive functional modules are powered-down during the execution of the application.

3.2.2.2 Output Register Delay Array

As explained in Section 3.1.3, there is no arbitration logic in the interconnect bus network. The time-sharing of the bus network is managed by the functional modules themselves. The bus network is time-shared mutually exclusively through controlled data transfers in predefined time-slots for each of the functional modules. These time-slots are determined through delay configuration values for each of the output registers.

Every output register of a functional module transfers data onto the bus network only after a specified number of clock cycles. This delay is controlled through the *delay* configuration register for each of the output registers in a functional module. Therefore, when programming an application, one must find the delay values for each of the output registers. The delay values are chosen in such a way that there is no conflict in the usage of bus network. The delay values are automatically identified through a scheduling algorithm which is explained in Chapter 4.

This processor architecture supports reuse of a functional module within a single schedule period. The delay values for the output registers can vary every time when a module is reused. Therefore, for each of the output registers there is an array of delay configuration registers. The size of this array determines the number of times a module can be reused. This size is parameterized through the *MAX_REUSE* parameter of the module design. A particular delay value is chosen from this array through a counter index *(config_index)* controlled by a wrapper configuration counter block.

The register width of *delay* configuration with the current packet structure is $five$ bits using the full $CD_{4..0}$ for configuration. Therefore, the maximum delay value supported is 31.

3.2.2.3 Output Register Destination Array

This configuration indicates the destination address, to which the data in each of the output registers of a module is transferred. Similar to the *delay* configuration, the *destination* configuration is an array of registers. The size of this array also is controlled by the *MAX_REUSE* parameter. Thus, each delay configuration will have a corresponding destination configuration and they will be used in tandem. The array of configuration registers share a common address and are filled in a round-robin fashion.

3.2.2.4 Num_used

The *MAX_REUSE* parameter specifies the maximum number of times a module can be reused in a single schedule period. Nevertheless, for a particular application, a module need not be reused *MAX_REUSE* times. Therefore, the actual number of times a module is reused in a single schedule period for a particular application is stored as a configuration in the *Num_used* wrapper configuration register. The wrapper configuration data is generated automatically through the mapping and scheduling algorithm discussed in Chapter 4.

3.2.3 Internal Module

In all the functional modules, the internal module is the core functional component. Therefore, the implementation of the internal module varies for each of the functional modules. Figure 3.9 shows a generic template structure for an internal module block.

When the input registers receive all of their input data, a common *input_reg_file_rdy* signal is generated. When the internal module reads the *input_reg_file_rdy* signal, it copies the data in the input register file and performs the actual function of the module to calculate the output. The number of execution cycles of the internal module can vary by the type of functional modules. Nevertheless, the number of execution cycles of the internal module of a particular

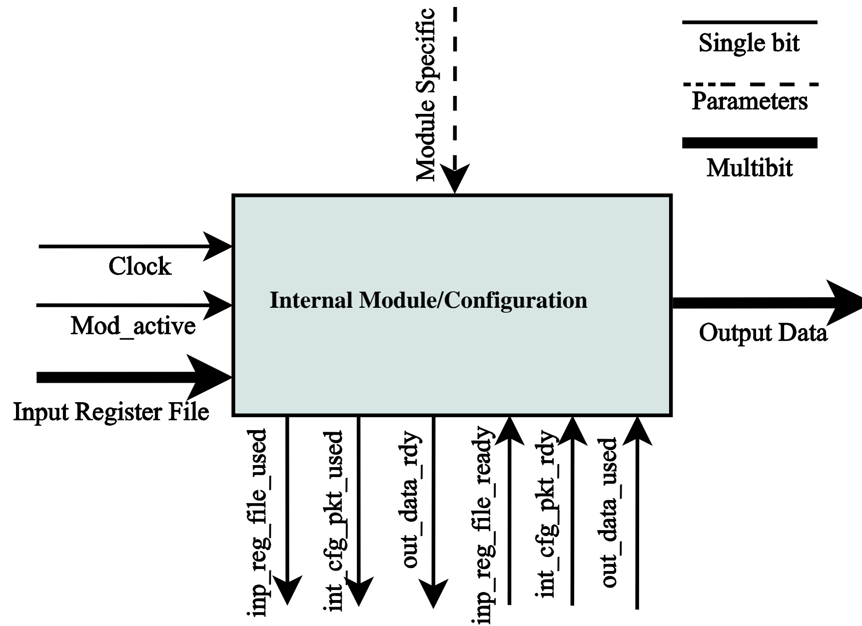

Figure 3.9: Internal Module Top Level Design

functional module is fixed. This allows the static deterministic scheduling of applications.

3.2.3.1 Internal Configuration Registers

Apart from the wrapper configuration registers, there are some module specific configuration registers that are called internal configuration registers. While the wrapper configuration registers control the time-sharing and destination address of the output registers, the internal configuration registers allows for module specific internal control. For example, the constant value of a constant generator module is specific to the application behavior. Therefore, the constant value is stored in an internal configuration register within the internal module.

The internal configurations are specified by the application graph node itself from which the configuration packets are automatically generated through the automation framework designed for the architecture. If the module is reused, each instance will have a different context based on the node configurations. Therefore, similar to the wrapper configurations such as delay-value and destination address, the internal configurations are also an array of

registers. For example, if in an application program there are three different constant values, the constant generator module will be reused three times with three different values stored in its internal configuration array.

3.2.4 Output Register(s)

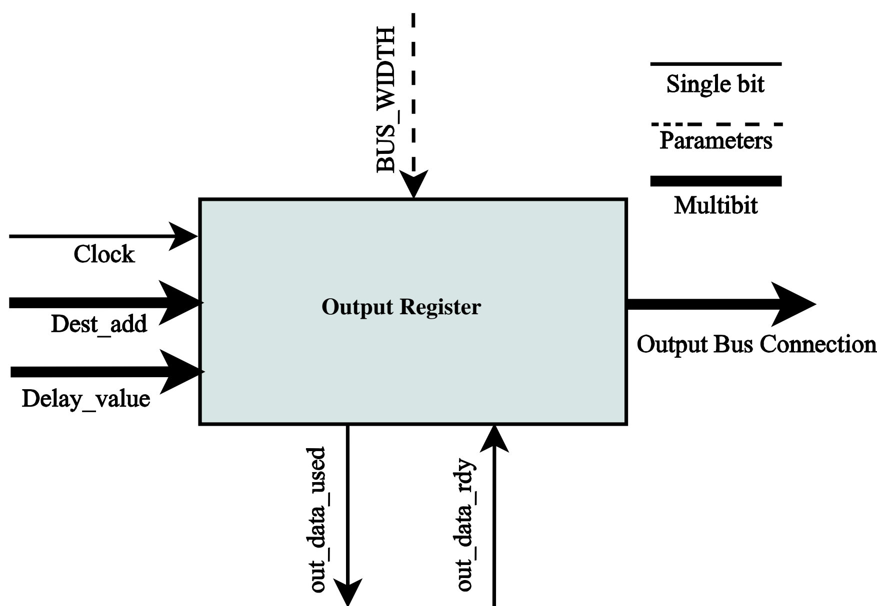

Figure 3.10: Output Register Top Level Design

The overall structure of an output register is shown in Figure 3.10. At the end of the internal module execution, the internal module generates the *out_data_ready* signal. The output register reads the output data from the internal module and the destination address from the wrapper configuration and forms an output packet. Each output register can have a different destination address stored in the corresponding wrapper configuration register. Similarly, the output registers can have different delay values. The output register holds the data for *delay* number of cycles and then transfers the data onto the output bus.

3.3 CAM-based Generic State-Machine Design

Many of the embedded applications have a control-flow structure, whereby the application execution branches out based on comparison of physical values with predefined thresholds. To implement such a structure in this architecture, there is a need for an SME. However, to retain the static schedulability feature of the architecture, this SME behavior should be deterministic with a constant number of execution cycles. Therefore, to have an ISA approach to implement the SME is not a solution. To this end, a CAM-based generic SME is designed as an internal module with a constant execution cycle. This is the only module that stores a “state”. Therefore, the power-aware design should have additional logic to retain the “state” before switching of the module.

The example CAM-based SME design takes two inputs. Based on the input data values, the internal module of the SME generates a *seven* bit context as illustrated in Figure 3.11.

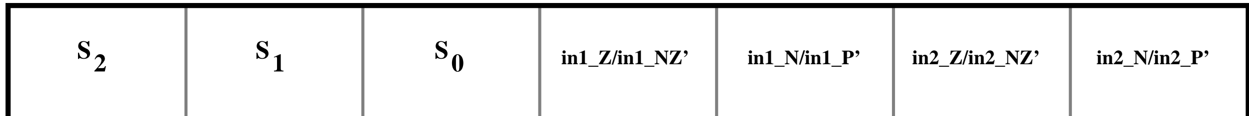

Figure 3.11: Index Formation in SME

The description of each of the *seven* bits in the index is explained below.

$S_{2..0}$

These *three* bits indicate the current state of the SME.

$in1_Z/in1_NZ'$

If this bit is *one*, it indicates that the input data1 of the SME has a value *zero*.

$in1_N/in1_P'$

If this bit is *one*, it indicates that the input data1 of the SME has a *negative* value.

$in2_Z/in2_NZ'$

If this bit is *one*, it indicates that the input data2 of the SME has a value *zero*.

$in2_N/in2_P'$

If this bit is *one*, it indicates that the input data2 of the SME has a *negative* value.

Based on the index, the SME gives the corresponding predefined data as the output. The *eleven*-bit output data consists of *eight* bits of output data and *three* bits of next-state value. With the implemented context formation strategy, state transitions controlled through threshold comparisons are easily implemented. With careful programming of the CAM, more general control flow strategies can also be implemented, as implemented in the *two* bit multiplier application (see Appendix A).

3.4 Sample Timing Diagram

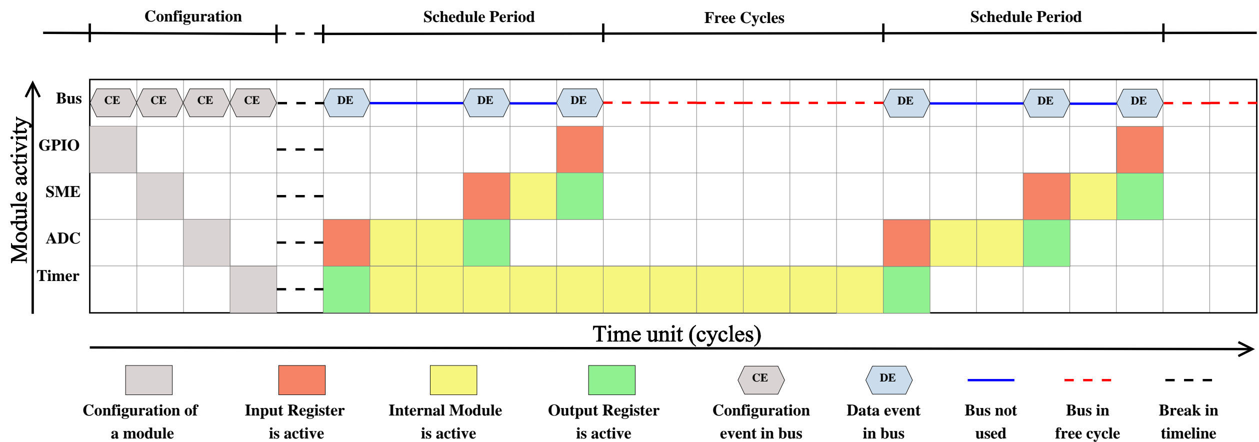

Figure 3.12: Sample Timing Diagram

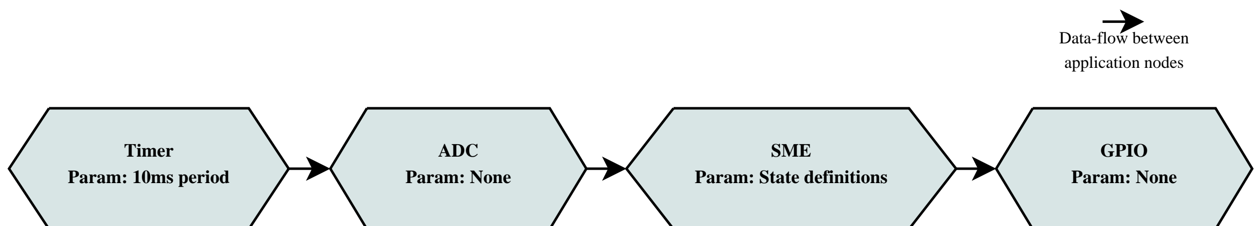

Figure 3.13: Application Graph for Thermostat Application

Figure 3.12 shows a sample timing diagram for a simple thermostat application (Figure 3.13). The application consists of an ADC sampling a sensor value under the control of a timer. The sensor value is sent to an SME which controls an actuator through a GPIO based on the sensor value.

The timing diagram starts with a *"Configuration"* stage, during which all the functional modules are configured for the application behavior. The timer module is configured last as it is the source module. The *"Configuration"* stage is followed by the *"Schedule Period"*, during which the actual application execution happens. The timer transfers the timer event to the ADC module. During this time, the output register of timer and input register of ADC operate concurrently. The ADC samples the sensor value and transfers it to the SME. Finally, the SME transfers its output to the GPIO. The GPIO, being a sink module, does not generate any bus output,but instead directly controls a connected actuator. As shown in the Figure 3.12, the execution times of the internal modules of various functional modules differ. The *"Schedule Period"* is followed by *"Free Cycle Period"* which is controlled by the configuration of input FIFO. During this period, the application can be reprogrammed through the network. The *"Schedule Period"* and *"Free Cycle Period"* execute alternately till the processor is powered down or reconfigured. The timer period is equal to the sum of *"Schedule Period"* and *"Free Cycle Period"*.

In this application, only one output was ready at any point of time. Therefore, the delay configuration of all the output registers is *zero*.

Chapter 4

Automation Framework Design

This chapter describes the design of an automation framework for the processor architecture. The automation framework is designed to achieve rapid prototyping of architecture instances, simpler modeling of applications, automated generation of configurations, and automated validation of the implementations. As explained in Chapter 3, the programming of the processor is done through simple configurations that manage the behavior of the functional modules. The internal configurations are application-specific and are given along with the application program itself. In contrast, the wrapper configurations reflect the information flow of the application on the architecture. Therefore, the main focus of the automation framework is to simplify the programming of the processor architecture by automatically generating these configurations.

This chapter discusses the overall outline of the automation framework. This is followed by the explanation of the object-oriented design of the framework. Finally, the implementation of individual components of the framework is explained in detail.

4.1 Automation Framework Outline

This section gives a high-level overview of the automation framework designed for the architecture. Figure 4.1 illustrates the top-level flow of the components in the framework.

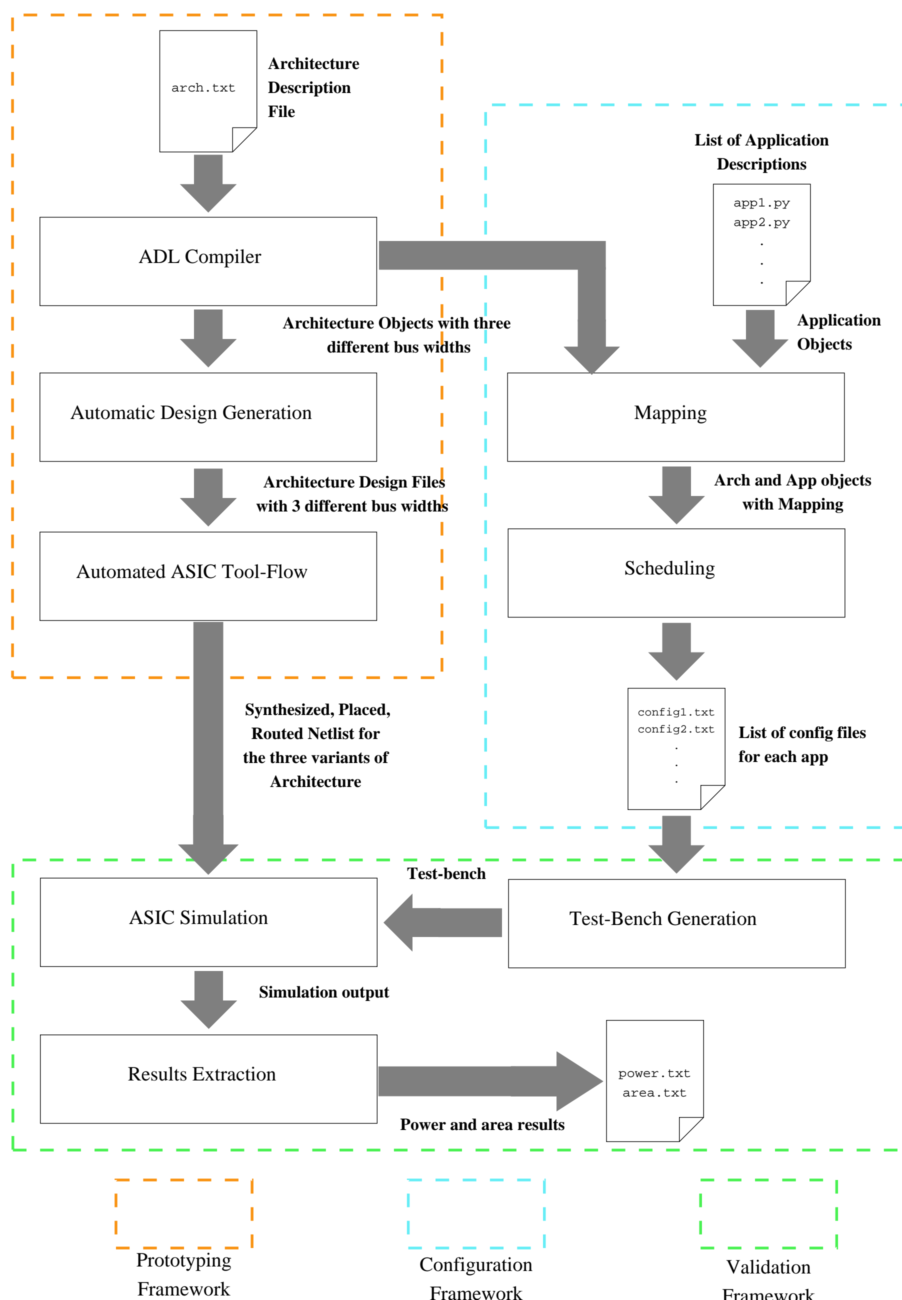

Figure 4.1: Framework Overview

The automation framework consists of three main components, a prototyping framework, a configuration framework, and a validation framework. The prototyping framework seeks to support rapid prototyping of architecture instances from high-level architecture descriptions. The configuration framework automates the generation of configurations for an application/architecture configuration using the architecture and application models. The validation framework automates the test generation to speedup the validation of an application/architecture combination.

The framework design is implemented using the Python programming language in an object-oriented manner [26]. The inputs to the framework are a base hardware architecture and a collection of target applications. The architecture and applications are represented in a simple framework-defined representation as explained in Sections 4.2.1 and 4.2.2.

The architecture input is given as a high level description using a simple ADL. The hardware design files for these architectures are automatically generated using the architecture Python objects. Then, the hardware designs are synthesized, placed, and routed through an automated ASIC tool-flow chain.

The input applications are given as high-level Python objects. All the input applications are mapped onto the three variants of the input architecture. During the mapping phase, the high-level application nodes are mapped to matching coarse-grained modules in the architecture. These mapped objects are used by a scheduler, which schedules the applications on the architecture, automatically generating the configurations to program the applications. Test-bench generation automates the generation of test-input packets for each the application/architecture combination. All applications are simulated on each of the architecture variants using an automated simulation framework. The power and area details are extracted automatically from the simulation output.

4.2 Architecture and Application Representation

As explained in Chapter 2, this thesis aims to achieve an MDE based design for applications. To this end, the components of applications and architectures are defined as a class of Python objects. With this design, applications and architectures are modeled simply by instantiating the corresponding Python objects. The Unified Modeling Language (UML) representations of the application and architecture class are given in Appendix C with a high-level overview of the objects.

4.2.1 Architecture Representation

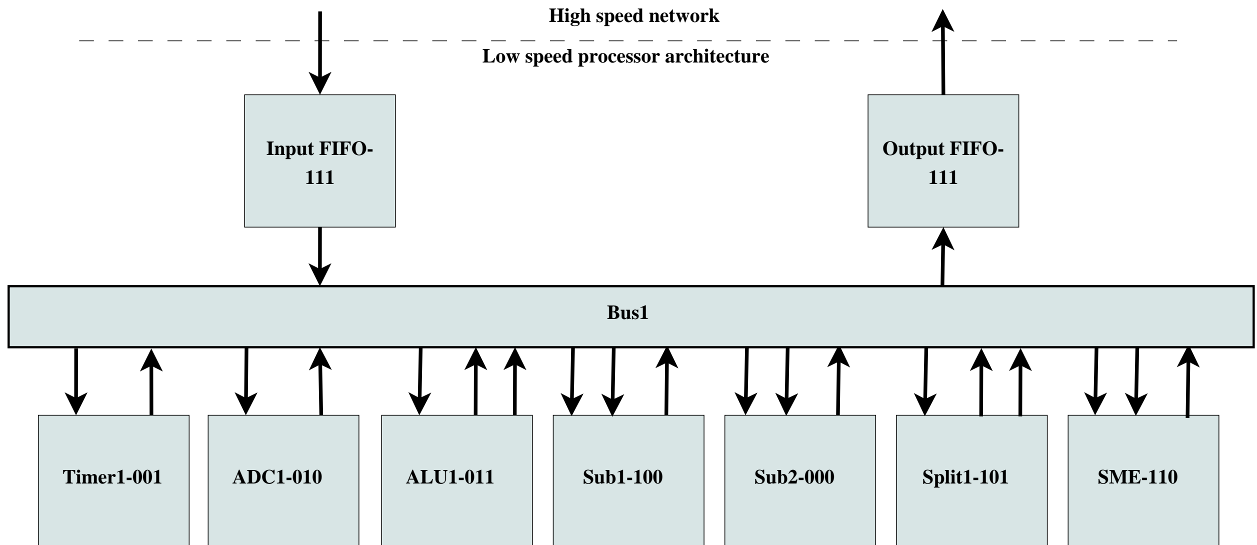

Figure 4.2: Architecture Model for Architecture-I

The architecture is represented as a graph showing the bus connections between the hardware modules, given by $Arch_G = (V_{arch}, E_{arch})$, where V_{arch} is the set of all input and output registers of all hardware modules and E_{arch} is the set of buses in the architecture. The edge (bus) in the architecture graph is defined as a connection between an output register of a module to the input register of a module. The edges in the architecture graph are bidirectional as the data transfer in the bus happens in both the directions.

4.2.1.1 Architecture Description Language

A sample architecture graph is given in Figure 4.2. Representing an architecture with the defined python object structure is challenging. Therefore, an ADL is defined to describe architecture instances in a simpler way. The ADL compiler is defined using Python Lex-Yacc (PLY), which is an open source python implementation of lex and yacc parser [27]. Using the defined ADL, the representation of architecture instances is straightforward. The ADL description for Architecture-I shown in Figure 4.2 given below.

```
1  parameters
2  BUS_WIDTH    4  ;
3  PACKET_WIDTH    12  ;
4  MAX_COUNT    3  ;
5  end parameters
6
7  buses
8  bus1    0  ;
9  end buses
10
11 components
12 fifo fifo1 7 1 inputs bus1 outputs bus1 localparams FIFO_DEPTH 8 PTR_WIDTH 3 ;
13 timer tmr1 1 1 inputs bus1 outputs bus1 localparams COUNTER_BITS 12 ;
14 adc_converter adc1 2 2 inputs bus1 outputs bus1 localparams RESOLUTION 8 ;
15 alu_plus_minus aluplusminus1 3 1 inputs bus1 outputs bus1 bus1 localparams ;
16 subtractor sub1 4 2 inputs bus1 bus1 outputs bus1 localparams ;
17 state_machine stmc1 6 2 inputs bus1 bus1 outputs bus1 localparams
       CONTEXT_WIDTH 7 DATA_WIDTH 7 ADDR_WIDTH 3 ;
18 splitter split1 5 2 inputs bus1 outputs bus1 bus1 localparams ;
19 subtractor sub2 0 2 inputs bus1 bus1 outputs bus1 localparams ;
20 end components
```

Listing 4.1: ADL File for Architecture-I

The ADL consists of three sections, parameters, buses, and components. The parameters section consists of architecture level parameters with their associated value. The buses section consists of a list of bus names with their associated bus id. The components section consists of a list of components with their associated details. The first keyword in the component declaration statement is the hardware module name (as in the Hardware Description Language (HDL) design file). The next operand in component declaration is the instance name for the component. This is followed by the module address and MAX_REUSE value for the particular component both of which are numbers. Next is the 'inputs' keyword followed by the list of input bus connections. Similarly, the 'outputs' keyword followed by the list of output bus list is given. Finally, the keyword 'localparams' followed by the list of local parameter name and value is given.

4.2.2 Application Representation

The application is represented as a DAG 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 nodes. An example application graph is shown in Figure 4.3.

The application nodes are high-level functions in the application. These are predefined application functions that can be mapped to the coarse-grained functional modules in the architecture. The edges indicates the data-flow between the application nodes. The application graph is modeled using the defined python classes. A sample application graph code is given in Appendix D.

4.3 Prototyping Framework

As shown in Figure 4.1, rapid prototyping of architecture instances has three steps, ADL compilation, automatic design generation, and tool-flow automation. This section gives a

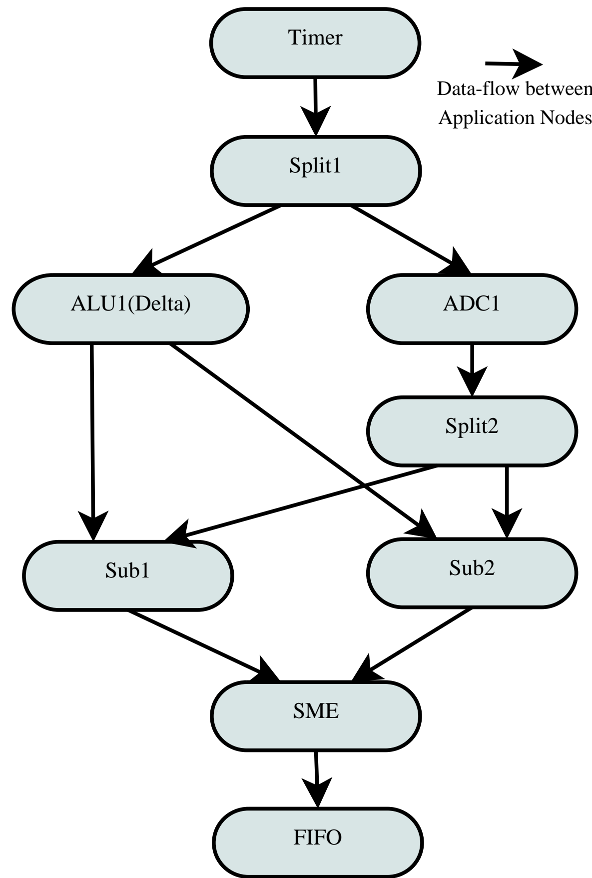

Figure 4.3: Application Graph for Application-I (Thermostat)

brief overview of each of these steps.

4.3.1 ADL Compilation

This step involves the transformation of architecture descriptions defined using the ADL to architecture python objects to be used by the framework. The ADL is fed as input to the compiler designed using PLY. The compiler transforms the ADL producing python objects for *three* variants of the given architecture with three different bus widths, *four*, *six*, and *twelve*. The bus widths are always factors of twelve to accommodate the current packet structure.

4.3.2 Automatic Design Generation

This step involves the automatic generation of design files for the given architecture instance. The framework has the template design files for all the functional modules in a particular folder. The architecture object has the list of file names associated with each of the functional modules. Thus, the design generation framework copies the necessary template files to the design folder based on the architecture object. Apart from these design files, the top-level HDL file is automatically generated for the given architecture.

4.3.3 Tool-Flow Automation

The generated design files are given as inputs to a tool-flow automation framework. This part of the framework consists of "tcl" script templates that are automatically customized for the given architecture design. The tool-flow consists of a Synopsys Design Compiler that synthesizes the design, and a Synopsys IC Compiler that places and routes the synthesized design [28]. Thus, the output of the tool-flow framework is synthesized, placed, and routed gate-level netlist for the architecture design variants [29].

4.4 Configuration Framework

The configuration framework is an important component of the automation framework automatically generating the configurations for an application/architecture combination. As shown in Figure 4.1, the configuration framework consists of two steps, mapping and scheduling. This section gives a detailed explanation of each of these steps.

4.4.1 Mapping

The framework consists of predefined hardware functional modules and application functional nodes. Based on the compatibility between the application node and the architecture module, all possible mappings are listed in a python object file. The inputs to the mapping algorithm are the application graph, the architecture graph, and this list of mappings.

4.4.1.1 Mapping Algorithm

The aim of the mapping algorithm is to map each of the application nodes to compatible modules in the architecture. Additionally, each data-flow edge in the application graph is mapped to a particular output register of the source module of the edge. This section describes the steps involved in the identification of mapping between the application and the architecture.

This is a simple backtracking algorithm that returns all possible mappings for the given application/architecture combination. The mapping algorithm randomly assigns a module from a list of available modules to an application node. Then, the validity of the mapping is verified. If the mapping is valid, then, the chosen mapping is retained, otherwise the next possible module is chosen in random and mapped. The validity of the mapping is verified by comparing the node type with the module type and by checking the availability of required input and output register connections. During each step, the list of available modules is updated based on resource sharing needs. This procedure is repeated for all the application nodes to find the mapping between the application nodes and hardware modules.

4.4.2 Scheduling

The scheduling framework identifies the delay values for each of the output registers in the architecture. The resulting delay values avoid bus and resource usage conflicts, also using

the three-stage pipeline within the functional modules. Additionally, the scheduler manages resource sharing of a functional module through cycle-accurate simulation of the application on the architecture. The inputs to the scheduling algorithm are the application graph, architecture graph, and the output of mapper.

At the highest level, the scheduler breaks up the scheduling problem using standard graph functions. The given application graph may have multiple sources, therefore, a set of simple subgraphs are obtained, with each of the subgraphs driven by a single source node. Furthermore, each subgraph is broken down into minimal subgraphs and zero-delay edges. The minimal subgraphs are parts of the graph that do not have parallel branches between them. Zero-edges are edges that connect these minimal subgraphs and their execution is independent of that of the minimal subgraphs. These edges do not have any edge parallel to them, therefore, they are assigned with zero delay value.

Each of the minimal subgraphs is independently scheduled. Edge delays decided by an invocation are fixed, and subsequent scheduler invocations are bound by those decisions.

The lower-level scheduler has two important components:

- The first component of the scheduler reads a representation of the entire application graph from the framework input. Some of the edges of the application graph are marked as ignored and others are marked as fixed. Ignored edges are the edges that are outside the minimal subgraph that is currently being scheduled. These edges will be ignored during the scheduling of the current subgraph. Edges that are marked as fixed have an already defined delay value from previous invocations and should not be assigned any other delay value.
- The second component of the scheduler does a search for delay values for all of the unmarked edges in the current minimal subgraph.

4.4.2.1 Scheduling Algorithm

This section describes the steps involved in the backtracking algorithm for the scheduling of minimal subgraphs. The scheduling algorithm assigns a delay-value in the *min_delay* to *max_delay* range, for each of the edges in the subgraph. If the assigned delay value does not result in bus conflict or resource usage conflict, then, the chosen value is retained, otherwise the next possible delay value is assigned. This procedure is repeated for all of the edges in the subgraph.

The verification of validity of the delay-value assignment involves the following steps. Time-line arrays that indicate the usage of buses, resources (internal module), input registers, and output registers are initialized with zero value. Then, the behavior of the hardware is replicated on a cycle-level accuracy, updating all the time-line arrays. Conflicts are identified during the updation of these arrays and the edge to be backtracked is determined. When conflicts occur, the delay value of the corresponding backtracked edge is increased by *one* and the whole process is repeated again.

With the mapping and scheduling output, all information regarding wrapper configuration registers such as mod_active, delay-values, destination addresses, num_used are available. The framework uses these values to create the configuration file that is given as input to the validation framework.

4.5 Validation Framework

The final component in the automation framework is the validation framework. The validation framework takes in the hardware-design and the automatic configuration as input. The validation framework has *three* steps, automatic test-bench generation, ASIC simulation, and results extraction.

4.5.1 Automatic Test-Bench Generation

The generated schedule for a particular application on an architecture has to be verified through simulation of the application. This simulation needs a test-bench to give configuration inputs and port data values in a timing controlled manner. This step of test-bench generation is automated in the framework. There are two test-bench related template Verilog files, namely, *read_config* and *gen_config*. The *read_config* module is used to read the configuration file that is generated by the scheduler. The *gen_config* file uses the data from the configuration file to generate the configuration packets in the architecture defined format. Apart from the wrapper configuration, the test-bench needs information about internal configuration and external port inputs. This information is available in the application model itself (as shown in Appendix D). The framework uses this information to generate a test_bench module that feeds the wrapper configurations, internal configurations, and test data to the architecture design for simulation.

4.5.2 Simulation and Results Extraction

The generated test-bench and the design netlist is used to simulate the application on the architecture. The application is simulated using the Synopsys VCS tool. The area and timing results are automatically extracted from the simulation output. Additionally, Synopsys PowerCompiler-PX is used to obtain power results using the simulation output.

Chapter 5

Validation and Results

This chapter discusses the methodologies that are followed in this thesis for validating the hardware implementation and functionality of the programming framework. Additionally, power consumption and area related results are obtained through experiments. These results are then analyzed for the energy-efficiency of the designed architecture.

5.1 Experimental Setup

The experimental setup mainly consists of the programming framework described in Chapter 4. An application graph and an architecture model are given as inputs to the programming framework. The framework automatically maps the application onto the architecture and schedules the application, automatically generating the configuration details for each of the hardware blocks in the architecture. The applications are executed at a clock frequency of 500 Khz. Moreover, variants of the given architecture are created with three different bus widths (4, 6, and 12). These architecture designs are fed as input to the tool-flow automation framework, which gives the power and area details for the application/architecture combination as an output.

5.2 Experimental Applications and Architectures

5.2.1 Applications

Three representative applications are used for the analysis of the architecture. Application-I is a simple thermostat application. This consists of an ADC, which samples a sensor value periodically based on a timer event. The output of the ADC is compared with a set temperature threshold. The difference is fed to a state-machine which controls the switch-on and switch-off of an actuator. The special feature of this application is the possibility of changing the temperature threshold through an asynchronous event from the network.

Application-II is a *two*-bit multiplier. This multiplier uses the basic "shift and add" approach to multiply the two operands. This application demands extensive resource sharing and hence is an ideal test application to validate resource-shared scheduling, pipelined implementation, and concurrent execution of multiple hardware modules.

Application-III is a hypothetical application that is used to demonstrate the configurability of a single processor architecture to execute multiple applications. Application graph for Application-I is shown in Figure 4.3. The other two application graphs are given in Appendix A.

5.2.2 Architectures

Three base architectures are used for the experiments. Architecture-I has a timer, an ADC, a special adder with asynchronous input, two subtracters, a state-machine, and a FIFO as coarse-grained modules. The thermostat application is tested on this architecture. Architecture-II is similar to Architecture-I, but has only one subtracter. The resource sharing aspect of Thermostat application is analyzed using this architecture. Architecture-III has a timer, an ADC, a splitter, a shifter, an adder, a state-machine, a constant-generator, and a FIFO as coarse-grained modules. The *two*-bit multiplier application and the hypo-

thetical application are tested on this architecture. Variations of these three architectures are created by changing the bus width.

Architecture-I is shown in Figure 4.2. The other two architectures are given in Appendix B.

5.3 Schedule Validation

Figures 5.1 and 5.2 illustrate the timing behavior of the *two*-bit multiplier application on Architecture-III that is obtained through framework-based simulations for the purpose of evaluating the schedule. The specific architecture variant in use has a bus width of 12. The schedule length for the given application/architecture combination is 47. The resource utilization of each module has information about the utilization of individual components of each module such as Input Registers(IRx), Internal Module(IM), and Output Registers(ORx). A '1' filled in the table indicates that the particular resource is in use, while a '0' indicates that the resource is free.

5.3.1 Validity of the Generated Schedule

This section analyzes the validity of the framework-generated schedules for different application/architecture combinations. Figures 5.1 and 5.2 illustrate the resource usage schedule for the Application-II/Architecture-III combination obtained through framework-based simulation. The framework-based simulation uses the configuration values that are automatically generated by the scheduling framework. Thus, there is a need to verify whether the same behavior is replicated in the actual hardware implementation. This is validated using the automatically generated test-bench, which is used by the VCS simulator providing a simulation output in the form of a timing diagram. The test-bench feeds the framework-generated delay-values as configurations to the hardware simulation. With the framework-generated delay-values, the hardware simulation showed a timing behavior without bus conflicts or

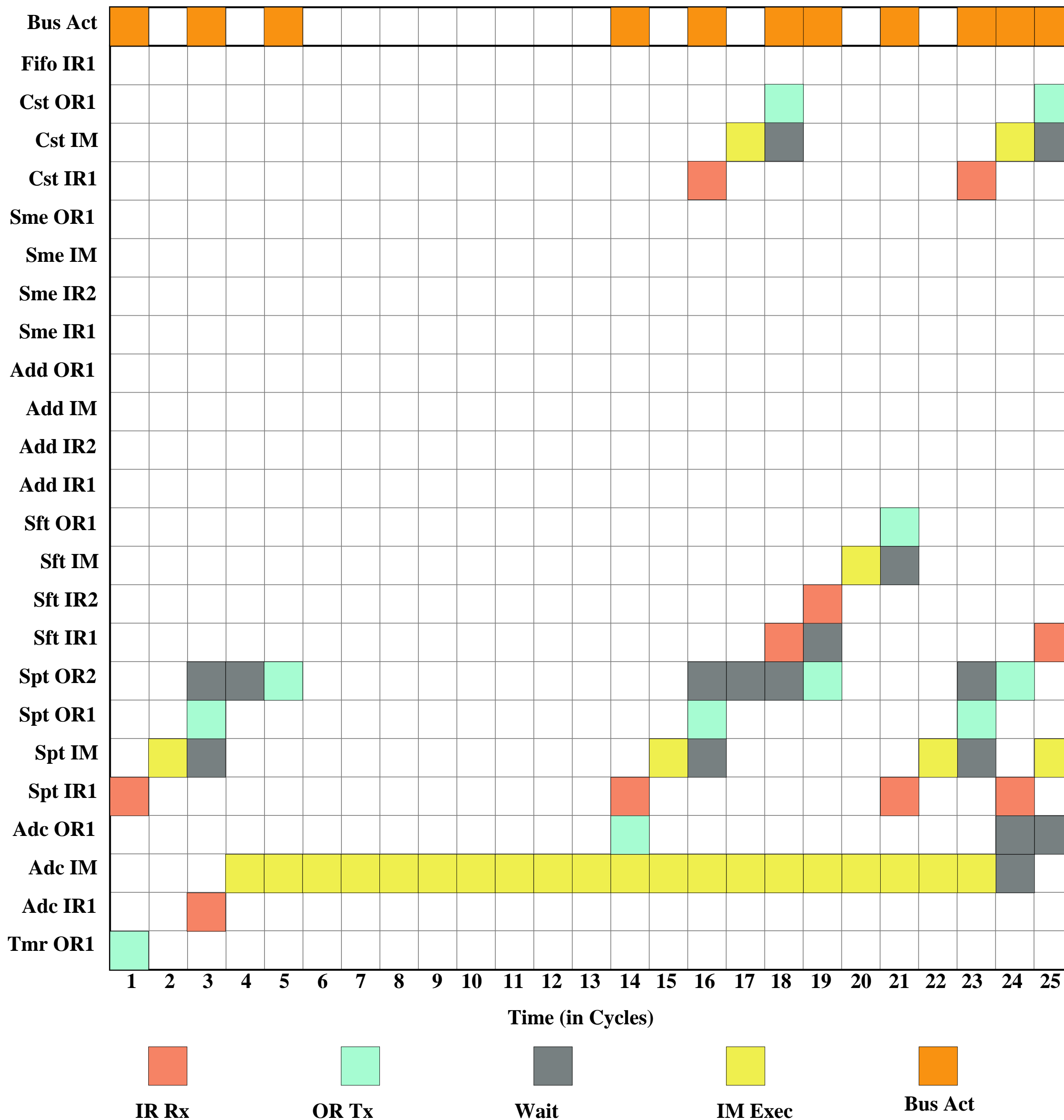

Figure 5.1: Schedule Diagram For Multiplier Application On Architecture-III (Part-1)

resource usage conflicts. This hardware-simulated timing diagram is compared on a cycle by cycle basis with the data in Figures 5.1 and 5.2 in order to validate the schedule. The framework-generated schedule is found to be matching the hardware-simulated timing dia-

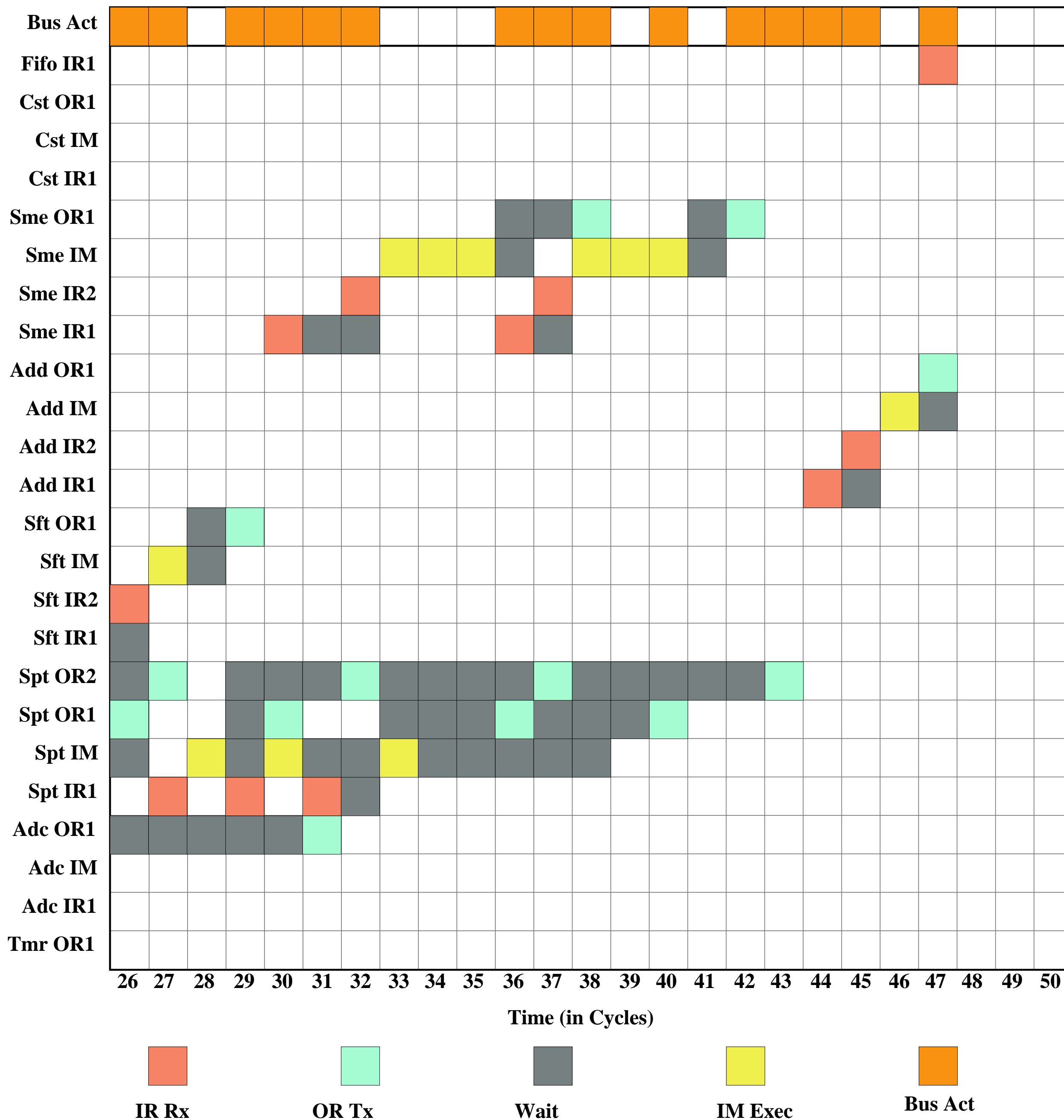

Figure 5.2: Schedule Diagram For Multiplier Application On Architecture-III (Part-2)

gram behavior for each cycle of executions. Similarly, the framework-generated schedules for other application/architecture combinations also are validated and found to be correct.

5.3.2 Resource Sharing, Pipelining, and Concurrency

This section concentrates on the analysis of the correctness of the implementation of resource sharing, pipelining and concurrent execution of resources in both the hardware implementation and the framework simulation. The thermostat application is executed on Architecture-I that does not necessitate any resource sharing. Without resource sharing every module in the hardware is used only once in a schedule period, leaving no scope for pipelining in the application. In contrast, the *two*-bit multiplier application execution involves extensive resource sharing. In particular, the splitter module is heavily reused. Therefore, the multiplier application is considered for the analysis of resource sharing and pipelining.

If a resource is reused, it will receive more than one set of inputs, one each for every execution. Therefore, resource sharing can be easily validated by the number of times a hardware module receives input in a single schedule period. The number of times a module receives inputs is equal to the number of times the resource is reused. In Figures 5.1 and 5.2, as per the IR1 row for splitter module, the splitter module receives input *seven* times. With the hardware simulation obtained using Synopsys VCS, there are no bus or resource-usage conflicts. Moreover, the data transfers between modules match the framework-generated schedule. Thus, both the hardware design and the programming framework function precisely with resource sharing.

As explained in the Chapter 3, each hardware module has a three-stage pipeline with input registers, internal module, and output registers. As given in Figure 5.2, IR1, IM, OR1, and OR2 of the splitter module are concurrently in use in cycles 31 and 32. This implies that the splitter module has one set of outputs ready to be transferred. During this period, the internal module has received another set of inputs and has calculated outputs that are ready to be transferred to the output registers. Simultaneously, the input registers have received a third set of inputs to be transferred to the internal module when ready. This schedule thus validates the proper functioning of the three-stage pipelining in the hardware implementation and also the framework simulation.

Concurrent operation of modules is easily explained through overlapping resource-usage of different modules in a given period of execution. For example, as per the data in Figure 5.1, the internal modules of ADC works concurrently with that of the splitter, shifter, and constant generator modules.

5.3.3 Configurability Analysis

This section demonstrates the configurability of the processor architecture. By design, the processor architecture executes different applications on a single hardware based on varying configurations. Table 5.1 shows the configuration values for certain modules for the execution of Application-II on Architecture-III. Similarly, Table 5.2 shows the configuration values for certain modules for the execution of Application-III on Architecture-III. From the tables, it is shown that, with two different configurations, a single hardware architecture (Architecture-III) is calibrated to application-specific behavior executing two different applications (Application-II and III).

Table 5.1: Configuration Values for Application-II on Architecure-III

Module	Delay	Destination	Num_used	Internal Configuration
ADC	0,7	5,5	2	-
Constant	0,0	4,4	2	6,1
Adder	0	7	1	-

Table 5.2: Configuration Values for Application-III on Architecure-III

Module	Delay	Destination	Num_used	Internal Configuration
ADC	0,0	5,4	2	-
Constant	4	4	1	5
Adder	-	-	-	-

5.4 Energy Usage Analysis

This section presents the analysis of the energy usage of the designed architecture for different applications. For all these analyses a non-power-aware design of the architecture is used. Instances of the architecture are created for analyzing the effects of resource sharing and change in bus width on the energy usage. Finally, a comparison between the energy usage of the designed architecture and that of a PIC micro-controller is presented, making a case for the energy-efficiency of the architecture for the target applications. In the subsequent sections, energy usage of the designed processor architecture is calculated by the following formula:

$$
\begin{aligned}
E_{sched} &= (P_{avg}) * \tau * N, \; where, \\
& E_{sched} \; is \; the \; energy - usage \; estimation \; for \; a \; single \; schedule \; period \; in \; Joules, \\
& P_{avg} \; is \; the \; average \; power \; consumption \; for \; a \; schedule \; period \; in \; Watts, \\
& \tau \; is \; the \; processor \; clock \; period \; in \; seconds, \\
& N \; is \; the \; number \; of \; clock \; cycles \; for \; the \; execution \; of \; a \; single \; schedule \; period.
\end{aligned}
$$

5.4.1 Energy Usage vs. Bus Width

Among all other factors considered here, bus width of the architecture has the largest impact on the schedule length. The data-flow nature of the architecture necessitates frequent data transfer between the coarse-grained modules. Therefore, the change in bus width results in a wide variation in the schedule length, as illustrated by the Table 5.3.

In contrast to the perceptive behavior, an increase in bus width results in a decrease in total area of the architecture, as shown in Table 5.4. The input register counter and output register counter logic reduces with increase in bus width. Therefore, this decrease in logic

outweighs the effect of increase in bus width.

Table 5.3: Schedule Length vs. Bus Width

Architecture	Application	Bus Width	Schedule Length (cycles)
Architecture-I	Application-I	4	39
Architecture-I	Application-I	6	31
Architecture-I	Application-I	12	23
Architecture-III	Application-II	4	99
Architecture-III	Application-II	6	71
Architecture-III	Application-II	12	47

Table 5.4: Bus Width vs. Area

Architecture	Bus Width	Total area of architecture (μm^2)
Architecture-I	4	93791
Architecture-I	6	92886
Architecture-I	12	89332
Architecture-III	4	102999
Architecture-III	6	102389
Architecture-III	12	98480

With decrease in data transfer time (schedule length) and decrease in area, the increase in bus width often results in a decrease in energy usage for a particular application execution. The data presented in Table 5.5 reflects the expected behavior in the variation of energy usage for different application/architecture combinations with different bus widths.

5.4.2 Energy Usage of Template

As explained in Section 3.2, all of the functional modules of the architecture have a template wrapper structure apart from the functionality-specific internal module. Therefore, the energy consumption of this template is an important parameter to be analyzed. This

Table 5.5: Bus Width vs. Energy Usage

Architecture	Application	Bus Width	Energy usage for a schedule period (nJ)
Architecture-I	Application-I	4	21.9
Architecture-I	Application-I	6	17.2
Architecture-I	Application-I	12	12.4
Architecture-III	Application-II	4	60.5
Architecture-III	Application-II	6	43.0
Architecture-III	Application-II	12	27.7

section analyzes the variation of the energy cost of the template with different bus widths. Additionally, the energy cost comparison between the template and the internal module of different functional modules are presented.

Table 5.6 shows the area and energy cost for the different components of the template. The data in the table is provided for architectures with bus width 4. The energy cost per cycle is calculated by taking the average of the energy usage of the architecture for a single schedule period of the application. The wrapper configuration and the wrapper counter blocks have *MAX_REUSE* parameter set as *one*. This is to ensure that the analysis of the template is done using the basic skeleton structure of the template. The increase in area and energy cost of the wrapper configuration and the wrapper counter blocks, with increase in the *MAX_REUSE* parameter, are analyzed separately in Section 5.4.3.

Table 5.6: Area and Energy Cost of Template Components

Template Component	Area (μm^2)	Energy cost per cycle (pJ)
Input Register1	1781.5	10.5
Input Register2	1781.5	9.8
Output Register	1828.1	13.7
Wrapper Configuration	1114.6	6.7
Wrapper Counter	76.5	0.4

As illustrated in Table 5.6, energy cost for the input register1 is higher than that of the input register2 (area of both the registers are same). This is because the input register2 is enabled only after the input register1 has finished receiving a packet. Therefore, the input register2 operates for a lesser period than the input register1. The data given in Table 5.6 is used to calculate the area and energy cost of *three* different template structures implemented in the functional modules as shown in Table 5.7.

Table 5.7: Area and Energy Cost of Three Different Templates

Template	Components	Area (μm^2)	Energy cost per cycle (pJ)
Template-I	one input, one output	4800.7	31.3
Template-II	one input, two outputs	6628.8	45
Template-III	two inputs, one output	6582.2	41.1

Table 5.8 shows the area and energy cost comparison between the template and the internal module of various functional modules. The area and energy cost of the internal module is lesser than that of the template for most of the functional modules. Nevertheless, for the SME module, the area and energy cost of the internal module is higher than that of the template structure. This is because the SME is designed using a CAM to have a deterministic number of execution cycles. The CAM itself occupies 6614.6 μm^2 of the total area of SME internal module (8537.7 μm^2). Similarly, the energy cost of the CAM alone is 38.6 pJ out of the total energy cost of SME internal module (49.4 pJ).

Table 5.9 shows the change in area of the template with different bus widths. Similarly, Table 5.10 shows the change in energy cost of the template with different bus widths. With increase in bus width, the area and energy cost of the template reduces. This is because of the decrease in packet formation logic in the input and the output register with increase in bus width. The change in bus width results in change of the area and energy cost of the input and output register components only. The area and energy cost of the wrapper configuration and the wrapper counter blocks are not affected by the change in bus width.

Table 5.8: Template vs. Internal Module

Functional Module	Template	Template Area (μm^2)	IM Area (μm^2)	Template Energy cost per cycle (pJ)	IM Energy cost per cycle (pJ)
ADC	Template-I	4800.7	1926.2	31.3	10.4
Splitter	Template-II	6628.8	1650.7	45	9.5
Adder	Template-III	6582.2	1221	41.1	6.5
Constant	Template-I	4800.7	1714.4	31.3	10.2
Shifter	Template-III	6582.2	2532.4	41.1	17.1
SME	Template-III	6582.2	8537.7	41.1	49.4

Table 5.9: Template-I Area vs. Bus Width

Bus Width	IR Area (μm^2)	OR Area (μm^2)	Template Area (μm^2)
4	1781.5	1828.1	4800.7
6	1682.0	1844.3	4717.4
12	1409.6	1754.6	4355.3

As shown in Tables 5.9 and 5.10, the area and energy cost of the input register decreases with increasing bus width. However, the output register does not show a similar pattern. The area and energy cost of the output register increases when the bus width is changed from $four$ to *six*. This is because the cells that drive the bus are larger in size compared to the input buffers in the input registers. Therefore, the increase in drive cells area is more than the decrease in packet formation logic, resulting in an increased area and energy cost. Nevertheless, when the bus width is changed from *six* to *twelve*, the decrease in packet formation logic dominates the increase in drive cell area, resulting in a decreased area and energy cost.

Table 5.10: Template-I Energy Cost vs. Bus Width

Bus Width	IR1 Energy Cost per cycle (pJ)	OR Energy Cost per cycle (pJ)	Template Energy Cost per cycle (pJ)
4	10.5	13.7	31.3
6	9.4	14.2	30.7
12	8.5	13.6	29.2

5.4.3 Energy Usage vs. Resource Sharing

This section analyses the effect of resource sharing on energy usage of the architecture. Typically, when a single resource is shared instead of using multiple resources, the schedule length increases. This is due to the fact that the data transfers might be delayed due to the shared resource being in use at a given instant.

With the restriction on the number of hardware modules in the current version of the architecture, a *two*-bit multiplier design without resource sharing is not feasible. Therefore, the thermostat application is used for this analysis. The thermostat application has two subtracter nodes. This application is scheduled on two different architectures, one with a single subtracter module and the other with two subtracter modules. The application is scheduled on both the architectures with three different bus widths. Table 5.5 illustrates the explained behavior by summarizing the energy usage and schedule length for all the schedules.

As illustrated in Table 5.11, there is an increase in the schedule length for the application execution with resource sharing. Nevertheless, resource sharing has resulted in energy savings of $\approx 1.4\%$ for bus width $four$ and $\approx 2.4\%$ for bus width six. In contrast, the energy usage for the architecture with bus width *twelve* has increased by $\approx 0.8\%$. This is because the corresponding increase in schedule length is higher for bus width *twelve* ($\approx 8.7\%$) compared to that of bus width *six* ($\approx 6.5\%$) and bus width $four$ ($\approx 7.7\%$). The architecture with bus width *six* has the least percentage increase in schedule length, thus offering the highest percentage increase in energy savings. However, the extent of increase in the schedule length

Table 5.11: Resource Sharing vs. Energy Usage

Resource Sharing	Bus Width	Schedule Length	Energy usage for a schedule period (nJ)
No Resource sharing	4	39	21.9
No Resource sharing	6	31	17.2
No Resource sharing	12	23	12.4
Subtracter Resource shared	4	42	21.6
Subtracter Resource shared	6	33	16.8
Subtracter Resource shared	12	25	12.5

for different bus widths is an application specific behavior. Therefore, while identifying a suitable architecture for a set of target applications, the resource sharing depth should be analyzed.

To understand the effect of resource sharing on the energy usage and area of the architecture, the increase in energy cost and area of the template with increase in resource sharing depth (*MAX_REUSE*) is analyzed. As explained in Section 5.4.2, the change in bus width does not have any effect on the area and energy cost of the wrapper configuration and wrapper counter block. Therefore, it is enough to analyze the effect of changing *MAX_REUSE* on the wrapper configuration and wrapper counter block for a single bus width. Table 5.12 shows the change in area and energy cost of the wrapper configuration and the wrapper counter blocks with change in the resource sharing depth.

As shown in Table 5.12, the wrapper configuration area (and energy cost) increases rapidly with increase in the resource sharing depth. The wrapper counter block increases in area only when the wrapper counter width increases (when the resource sharing depth changes from *two* to *four*, *four* to *six*, and *eight* to *ten*). Therefore, resource sharing should be used appropriately based on the functional module usage and the target application needs.

Table 5.12: Resource Sharing vs. Wrapper Configuration

Resource sharing depth	Wrapper configuration area (μm^2)	Wrapper counter area (μm^2)	Energy cost per cycle for Wrapper configuration (pJ)	Energy cost per cycle for Wrapper counter (pJ)
1	1114.6	76.5	6.7	0.4
2	1975.9	76.5	11.8	0.4
4	3751.0	195.0	21.0	1.3
6	5551.0	311.4	29.4	2.0
8	7233.8	311.4	40.2	2.0
10	8931.0	398.0	53.2	2.6

5.4.4 Energy Usage Comparison with PIC Family

The energy usage of the processor architecture is compared with that of a general purpose micro-controller (PIC12F675) [30]. The PIC12F675 is a simple PIC architecture with interrupt capability and an internal ADC. Thus, this architecture is comparable to the current version of the architecture designed in this thesis.

For the purpose of this energy efficiency comparison, the sample applications are implemented in MPLAB Integrated Development Environment (IDE) [31] using 'C' language and cross-compiled to the platform using HI-TECH C compiler [32]. The implementation is simulated on PIC12F675 using MPLAB SIM tool which is integrated within the MPLAB IDE. This simulation is used to obtain the number of instruction cycles required for the execution of the implemented applications. Using the manufacturer's published specifications and the number of instruction cycles for execution, energy estimates for the PIC12F675 micro-controller are obtained. The formula used for the energy estimation for the PIC implementation is given below:

$$
\begin{aligned}
E_{app} &= (V_{typ} * I_{typ}) * \tau * N, \ where, \\
& E_{app}\ is\ the\ energy-usage\ estimation\ for\ the\ application\ execution\ in\ Joules, \\
& V_{typ}\ is\ the\ typical\ value\ of\ supply\ voltage\ obtained\ from\ datasheet\ in\ Volts, \\
& I_{typ}\ is\ the\ typical\ value\ of\ supply\ current\ obtained\ from\ datasheet\ in\ Amperes, \\
& \tau\ is\ the\ processor\ clock\ period\ in\ seconds, \\
& N\ is\ the\ number\ of\ clock\ cycles\ for\ the\ execution\ of\ an\ application.
\end{aligned}
$$

In the above equation, N is the only variable which depends on the application. The other variables are fixed for a particular processor architecture and are obtained through datasheet. The ADC in PIC12F675 cannot operate on a voltage point below $2.5V$. Therefore, an operating point above $2.5V$ is chosen for energy estimation. The values of V_{typ} and I_{typ} for PIC12675 is found to be $3V$ and $190\mu A$ respectively, with the value for τ at $1\mu s$ [30]. Application-I and Application-II are implemented on PIC12F675 and found to take 66 and 98 instruction cycles respectively. The PIC implementations of these applications are given in Appendix E. These instruction cycles include the ISR overhead for context-switching. The number of instruction cycles to execute the ISR is found by using the *stopwatch* feature of the debugger by setting breakpoints at the start and the end of the ISR. The ISR context-switching overhead is found manually, by counting the number of instructions, using the *simulator trace* feature of the MPLAB IDE. In PIC12F675, each instruction cycle takes 4 oscillator cycles to complete. Therefore, the value of N for Application-I and Application-II are 264 and 392 respectively. These values are used to calculate the energy-usage estimates for each application on PIC12F675.

The energy usage values for the applications on the data-flow architecture is obtained through simulations. The simulation gives the average power consumption during a schedule period. The product of this with the schedule period gives the value of energy usage on the data-

flow architecture. Tables 5.13 and 5.14 show the energy usage comparison between processor architectures for Application-I and Application-II respectively.

Table 5.13: Energy Usage Comparison for Application-I

Architecture	Bus Width	Number of execution cycles	Time for execution (μs)	Energy usage (nJ)
PIC12F675	-	264	264	150.5
Architecture-I	12	23	46	12.4
Architecture-I	6	31	62	17.2
Architecture-I	4	39	78	21.9

Table 5.14: Energy Usage Comparison for Application-II

Architecture	Bus Width	Number of execution cycles	Time for execution (μs)	Energy usage (nJ)
PIC12F675	-	392	392	223.4
Architecture-III	12	47	94	27.7
Architecture-III	6	71	142	43.0
Architecture-III	4	99	198	60.5

From the energy usage values as shown in Table 5.13, it is evident that the designed data-flow processor Architecture-I (with bus width 12) is $\approx$ 12 times energy-efficient compared to the PIC micro-controller. Similarly, as shown in Table 5.14, Architecture-III (with bus width 12) is $\approx$ 8 times energy-efficient than the PIC micro-controller. There is significant energy-savings for architectures with other bus widths too. Moreover, even with a slower clock (500 Khz), compared to that of PIC12F675 (*one* Mhz), the application execution is faster in the designed architecture. This is because the number of instruction cycles for execution is more for the PIC micro-controller compared to that of the designed architecture.

Chapter 6

Conclusions and Future Work

This thesis shows the design of a novel processor architecture for a class of event-driven embedded applications. The design principles of this processor architecture are based on the target application requirements. The processor architecture is implemented using a data-flow approach to suit the event-driven nature of the applications. With a data-flow-based design, the processor architecture manages events natively without the use of interrupts or other separate event-handling mechanisms. In most of the embedded systems applications, there is a need for executing concurrent tasks. With the traditional ISA-based sequential processor approach, managing concurrency is complex. To this end, the processor architecture described in this thesis is designed as a non-ISA processor. The coarse-grained modules in the processor operate independently based on individual configurations. With this approach, concurrency is inherently managed in the processor architecture and does not require any effort from the user to implement concurrent tasks separately.

The target applications such as sensor-monitoring, operate on periodic events with stringent timing and energy-constraints. To guarantee the timing requirements, the processor is designed to have a deterministic timing behavior. During the "schedule period", the event-bus is time shared through predefined *delay* configurations. All external events pass through the event-FIFO and are transferred to the event-bus in the "free cycles". Thus, the entire

application is statically scheduled on the processor architecture resulting in a deterministic timing-behavior that guarantees correct operation with real-time bounds and power constraints. The overall processor architecture design seeks to achieve energy-efficiency. The coarse-grained modules in the architecture are implemented with a three-stage pipeline that increases the overall throughput. Also, the design of these modules can be optimized for energy-efficiency directly at the hardware level. Moreover, the handshake signals within the modules can be used by a hardware power manager to implement a power-aware design of the processor. With the implemented design, the processor architecture is shown to be energy efficient than the PIC12F675 general purpose micro-controller.

Apart from the target application requirements, the other important aspect considered in this thesis is reduced design complexity for the implementation of applications on this architecture. Towards this end, a simple configuration mechanism has been provided for programing the processor. To reduce the design complexity further, an automation framework is implemented. The automation framework is designed to achieve rapid prototyping of experimental processor architectures, automatic generation of configurations, and automatic validation of the implementations. Experimental architectures and applications are implemented using the framework demonstrating the usefulness of the framework. The implemented framework is object-oriented and this can be translated to an MDE approach with limited effort. Thus, the processor architecture design is shown to be a good target for MDE approach compared to the general purpose micro-controllers.

6.1 Future Work

This thesis demonstrates the benefits of the design principles used to implement the processor architecture and the automation framework. However, the current version of the architecture is limited by its packet structure, thus restricting the number of modules in the architecture to eight. This limits the class of applications that can be executed on this processor architecture.

Therefore, there is scope to increase the usefulness of this architecture to an extended set of applications by modifying the packet structure. Moreover, the current version of the architecture uses a single event-bus. Future implementations of the architecture can be extended to support multiple event-buses and more than two event-FIFOs.

Similarly, the current implementation of the automation framework includes a basic scheduling algorithm that aims to identify a valid schedule for a particular application/architecture combination. Despite the schedule being valid, it may not be optimal in terms of schedule length and energy-efficiency. Therefore, there is a need for a robust scheduling algorithm that aims to obtain the optimal schedule for a given application/architecture combination, which can be done as a future extension. The application modeling can be further simplified by integrating the automation framework with an MDE software such as Simulink.

Additionally, the automation framework is implemented with a limited architecture exploration capability. In the current implementation, three variants of a single base-architecture are created by varying the bus width. This architecture exploration space can be further expanded through modification of the number of buses, selection of modules, and resource sharing analysis.

Bibliography

[1] M. Weiser, "The Computer for the 21st Century," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 3, no. 3, pp. 3–11, 1999.

[2] Microchip Technologies. 8-bit PIC Microcontrollers. [Online]. Available: http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2119

[3] J. Hill, "A Software Architecture Supporting Networked Sensors," Master's thesis, University of California - Berkeley, 2000.

[4] V. Ekanayake, C. Kelly, IV, and R. Manohar, "An Ultra Low-Power Processor for Sensor Networks," *SIGARCH Comput. Archit. News*, vol. 32, no. 5, pp. 27–36, 2004.

[5] C. Gopalsamy, S. Park, R. Rajamanickam, and S. Jayaraman, "The Wearable Motherboard: The first generation of adaptive and responsive textile structures (ARTS) for medical applications," *Virtual Reality*, vol. 4, no. 3, pp. 152–168, September 1999. [Online]. Available: http://dx.doi.org/10.1007/BF01418152

[6] P. Grossman, "The LifeShirt: a multi-function ambulatory system monitoring health, disease, and medical intervention in the real world." *Studies in health technology and informatics*, vol. 108, pp. 133–141, 2004. [Online]. Available: http://view.ncbi.nlm.nih.gov/pubmed/15718639

[7] ElekTex, "ElekTex Textile Touchpads," http://www.eleksen.com/?page=products/elektexproducts/index.php.

[8] 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, no. 12, pp. 1993 – 1994, dec 2003.

[9] S. Park, K. Mackenzie, and S. Jayaraman, "The Wearable Motherboard: A Framework for Personalized Mobile Information Processing (PMIP)," in *DAC '02: Proceedings of the 39th annual Design Automation Conference*. New York, NY, USA: ACM, 2002, pp. 170–174.

[10] T. Martin, M. Jones, J. Edmison, and R. Shenoy, "Towards a design framework for wearable electronic textiles," in *ISWC '03: Proceedings of the 7th IEEE International Symposium on Wearable Computers*. Washington, DC, USA: IEEE Computer Society, 2003, p. 190.

[11] R. Shenoy, "Design of e-textiles for acoustic applications," Master's thesis, Virginia Tech, Blacksburg, 2003.

[12] J. B. Chong, "Activity Recognition Processing in a Self-Contained Wearable System," Master's thesis, Virginia Tech, Blacksburg, 2008.

[13] V. A. Kottapalli, A. S. Kiremidjian, J. P. Lynch, E. Carryer, T. W. Kenny, K. H. Law, and Y. Lei, "Two-tiered wireless sensor network architecture for structural health monitoring," in *Smart Structures and Materials 2003: Smart Systems and Nondestructive Evaluation for Civil Infrastructures*, S.-C. Liu, Ed., vol. 5057, no. 1. SPIE, 2003, pp. 8–19. [Online]. Available: http://link.aip.org/link/?PSI/5057/8/1

[14] M. T. Jones, T. L. Martin, and B. Sawyer, "An Architecture for Electronic Textiles," in *BodyNets '08: Proceedings of the ICST 3rd international conference on Body area networks*. ICST, Brussels, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, pp. 1–4.

[15] T. Martin, M. Jones, J. Chong, M. Quirk, K. Baumann, and L. Passauer, "Design and Implementation of an Electronic Textile Jumpsuit," *IEEE International Symposium on Wearable Computers*, vol. 0, pp. 157–158, 2009.

[16] eCos. eCos Embedded OS. [Online]. Available: http://ecos.sourceware.org/about.html

[17] Microsoft. Windows Embedded CE. [Online]. Available: http://www.microsoft.com/windowsembedded/en-us/products/windowsce/default.mspx

[18] Opensource. TinyOS. [Online]. Available: http://www.tinyos.net/scoop/

[19] M. Hempstead, N. Tripathi, P. Mauro, G. yeon Wei, and D. Brooks, "An ultra low power system architecture for sensor network applications," in *SIGARCH Comput. Archit. News*, 2005, p. 2005.

[20] V. Handziski, J. Polastre, J.-H. Hauer, C. Sharp, A. Wolisz, and D. Culler, "Flexible Hardware Abstraction for Wireless Sensor Networks," in *Proceeedings of the Second European Workshop on Wireless Sensor Networks, 2005.*, 31 2005, pp. 145 – 157.

[21] A. Schoofs, M. Aoun, P. v. d. Stok, J. Catalano, R. S. Oliver, and G. Fohler, "A Framework for Time-Controlled and Portable WSN Applications ," in *Institute for Computer Sciences, Social Informatics and Telecommunications Engineering.* Heidelberg, Berlin: Springer, 2010, pp. 126–144.

[22] C. Kelly, IV, V. Ekanayake, and R. Manohar, "SNAP: A Sensor-Network Asynchronous Processor," in *ASYNC '03: Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems.* Washington, DC, USA: IEEE Computer Society, 2003, p. 24.

[23] S. Kelem, B. Box, S. Wasson, R. Plunkett, J. Hassoun, and C. Phillips, "An Elemental Computing Architecture for SD Radio," 2007. [Online]. Available: http://www.sdrforum.org/pages/sdr07/Proceedings/Papers/1.5/1.5-4.pdf

[24] Mathworks. Simulink. [Online]. Available: http://www.mathworks.com/products/simulink/

[25] Reactive Systems. Reactis. [Online]. Available: http://www.reactive-systems.com/

[26] Open Source. Python. [Online]. Available: http://www.python.org/

[27] D. Beazley. Python Lex-Yacc. [Online]. Available: http://www.dabeaz.com/ply/

[28] "Synopsys Low-Power Solution," https://electronics.wesrch.com/User_images/Pdf/SE1_1191538495.pdf, 2007.

[29] R. P. Narayanaswamy, "Design of a Power-Aware Dataflow Processor Architecture," Master's thesis, Virginia Tech, Blacksburg, 2010, (to be published).

[30] Microchip Technologies. PIC12F675. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/41190G.pdf

[31] ——. MPLAB. [Online]. Available: http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en019469&part=SW007002

[32] ——. HITECH Compiler. [Online]. Available: http://www.htsoft.com/

[33] Open Source. NetworkX. [Online]. Available: http://networkx.lanl.gov/

Appendix A

Application Graphs

The application graphs for Application-II and Application-III as explained in Section 5.2.1 of Chapter 5 are given in this chapter. Figure A.1 shows the application graph for Application-II (Multiplier).

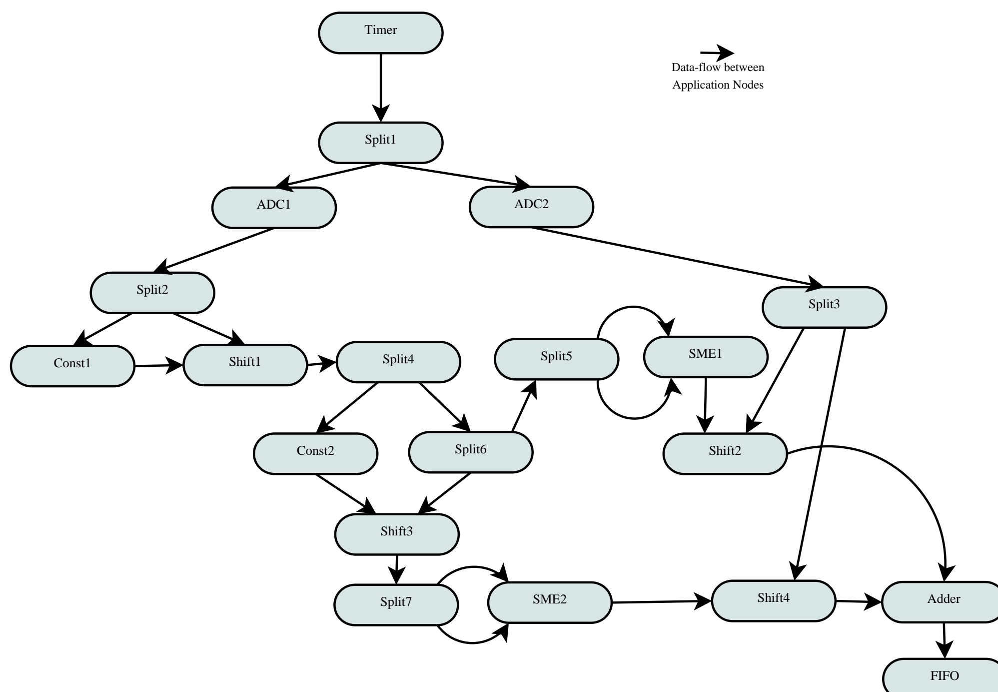

Figure A.1: Application Graph for Application-II (Multiplier)

Figure A.2 shows the application graph for Application-III (Hypothetical).

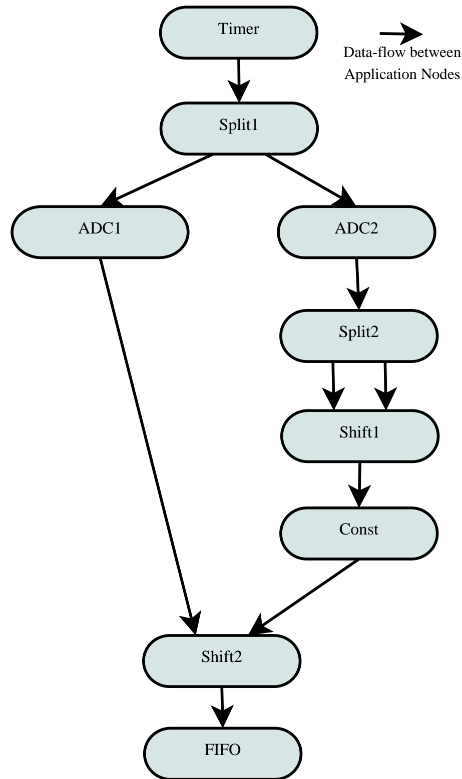

Figure A.2: Application Graph for Application-III (Hypothetical)

Appendix B

Test Architectures

The architecture diagrams for Architecture-II and Architecture-III as explained in Section 5.2.2 of Chapter 5 are given in this chapter. Figure B.1 shows the architecture diagram for Architecture-II.

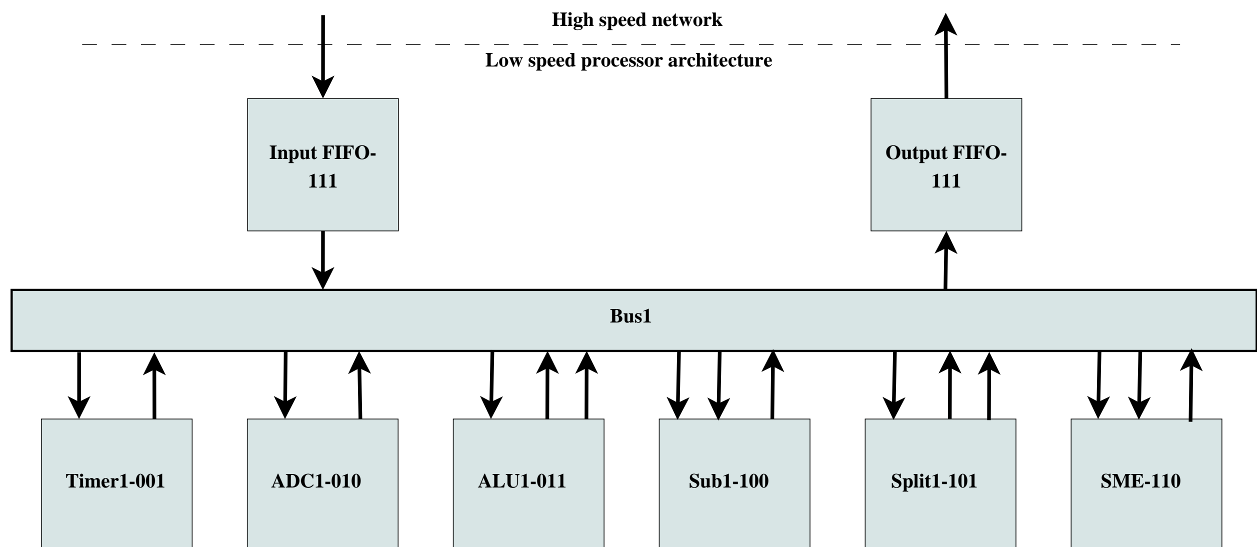

Figure B.1: Architecture Model for Architecture-II

Figure B.2 shows the architecture diagram for Architecture-III.

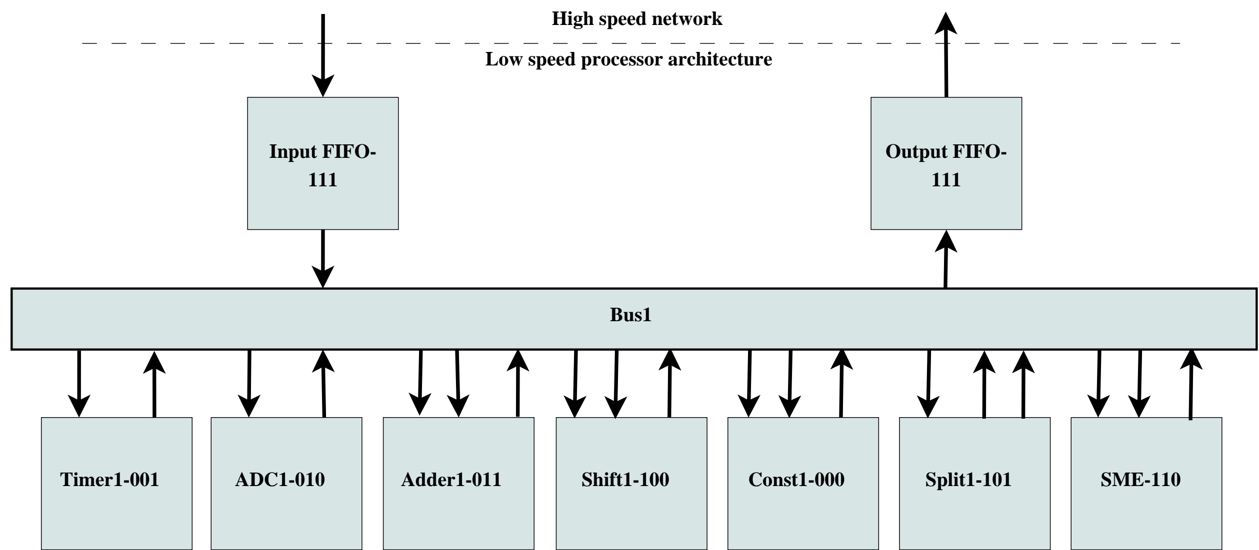

Figure B.2: Architecture Model for Architecture-III

Appendix C

UML Description of Application and Architecture Objects

This chapter gives the UML representation of the Application and Architecture Python classes used in the automation framework design discussed in Chapter 4.

C.1 Architecture Class

Figure C.1 shows a UML representation of the Architecture class. The architecture class aggregates a number of module instance objects. Each module instance object is associated with a module type class.

The architecture class consists of three main attributes as explained below.

- Clock : This indicates the clock period of the architecture
- Parameter_List : This is the list of architecture level parameters such as bus_width
- Bus_List : This is the list of bus names with associated bus_ids

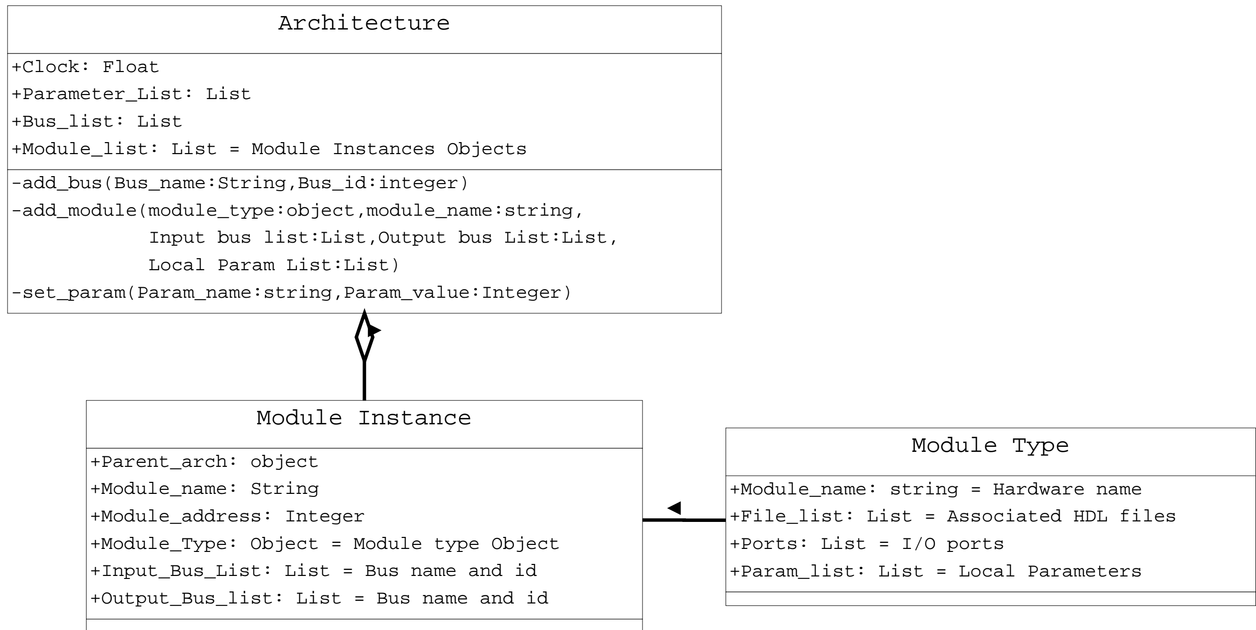

Figure C.1: Architecture Class UML Representation

- Module_List : This is the list of module instance objects used in this particular architecture

The methods associated with the architecture class are listed below.

- add_bus : This method adds a new bus to the architecture with an associated bus_name and bus_id
- add_module : This method adds a new module instance object to the architecture
- set_parameter : This method associates a value to the architecture level parameters

As shown in the UML representation of the Architecture Class, the architecture class aggregates a collection of module instance objects. The module instances are created using the add_module method of the architecture. The module instance class consists of the following attributes as explained below.

- Parent_Arch : This is the architecture object to which this module instance belongs

- Module_Name : This is the name of the module instance
- Module_Address : This is the hardware address of this module instance
- Module_Type : This is the module type object to which this instance is associated with
- Input_Bus_List : This is the list of input bus connections for this module instance
- Output_Bus_List : This is the list of output bus connections for this module instance

Each of the module instance has a hardware module type associated with it. The class of module_type has the following attributes.

- Module_Name : This is the name of the hardware module used by the module instance. This is same as the name of the module given in the HDL description
- File_List : This is a list of all the HDL files associated with the module design
- Ports : This is the list of additional local ports for the module apart from the general I/O bus connections. For instance, for the ADC module, "analog signal" port is a local port
- Param_List : This is a list of additional local parameters associated with the module apart from the general parameters such as MOD_ID, BUS_WIDTH, NUM_INPUTS, MAX_REUSE. For instance, an additional local parameter for ADC is RESOLUTION

C.2 Application Class

Figure C.2 shows a UML representation of the Application class. The App Graph class aggregates a set of application node objects. Each node object aggregates a collection of node_inputs, node_outputs and their associated edge objects.

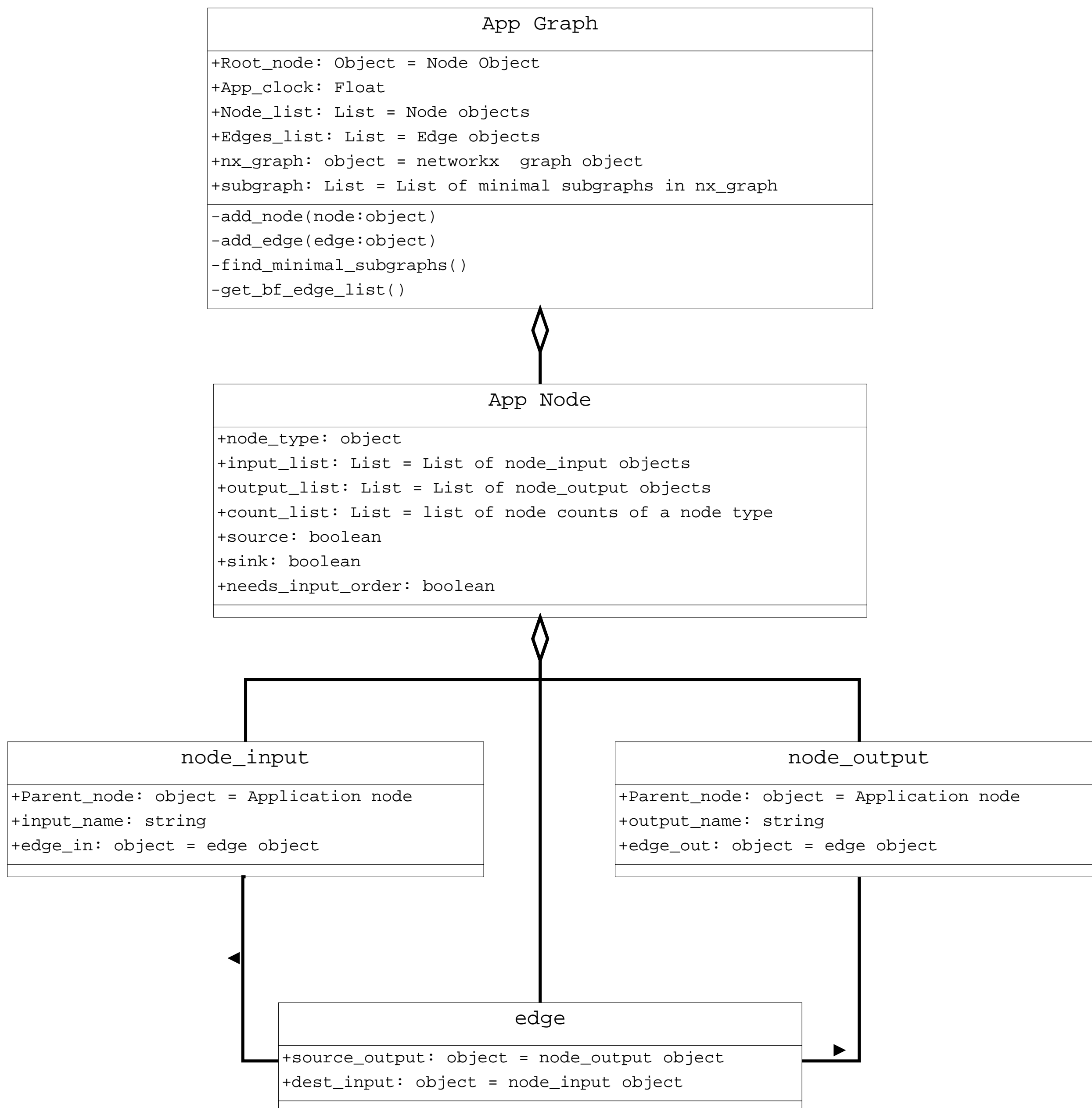

Figure C.2: Application Class UML Representation

The app graph class consists of the following attributes.

- Root_node : This is the root node of the application graph
- App_clock : This indicates the clock period in which the application executes

- Node_List : This is the list of node objects used in this particular application
- Edge_List : This is the list of edge objects indicating the data-flow in the application
- nx_graph : This is the Networkx graph object for the application graph. Networkx is an open source python package for complex network graph operations [33]
- sub_graph : This is the list of minimal subgraphs within the application graph

The app graph class consists of the following methods.

- add_node : This method adds an application node to the application graph
- add_edge : This method adds an edge between two application nodes
- find_minimal_subgraphs : This method returns a list of minimal subgraphs within the application graph
- get_bf_edge_list : This method returns a list of edges in the application graph in a breadth first order

The application graph consists of a collection of application nodes. The application node objects have the following attributes.

- node_type : This indicates the type of this particular application node such as adder, splitter etc ...
- input_list : This is the list of node_input objects for this particular node
- output_list : This is the list of node_output objects for this particular node
- source : This is the flag that indicates whether the application node is a source or not
- sink : This is the flag that indicates whether the application node is a sink or not

- needs_input_order : This is the flag that indicates whether the node needs to receive inputs in a specific order or not

Each application node has a collection of node_inputs and node_outputs with associated edge objects. These objects are used to specify the connections between the application nodes and their classes are self explanatory as shown in the figure C.2.

Appendix D

Sample Application Description

This chapter shows a sample application description. The application description given below is for Application-I (Thermostat).

Code listing 1 Sample Application Description

```
#!/usr/bin/python
from AppObjects import *
from AppGraph import AppGraph
#Creating application node instantiations with
#application-specific internal configurations
tmr = Timer(5.0)
tmr.set_config("period", 10800)#Time unit in ns
adc = ADC()
delta1 = Adder("plus_minus_delta")
delta1.set_config("delta_alupm", 9)
state = StateMachine()
state.set_config("context1", 0)
state.set_config("data1", 17)
fifo=Fifo()
#Giving the data-flow (edges) between the application nodes
#Each input and output will have associated tokens such as in1, A-B etc..
tmr.send(adc, "command", "expired")
adc.send(delta1, "command1", "sample_8")
delta1.send(state, "in1", "plusdel")
delta1.send(state, "in2", "minusdel")
state.send(fifo, "fifoin1", "out1")
#Creation of the App graph object
g = AppGraph()
g.set_clock(5.0)#Application clock
g.from_root(tmr)
#Test stimlus to be used by test-bench generator
test_data = {0: [(adc, "port_%s_analog_in", "00000000"),],
350000: [(adc, "port_%s_analog_in", "00100001"),
(fifo, "port_%s_nt_input_data", "011101100001"),
(fifo, "port_%s_nt_inp_data_busy", "1"),],}
```

Appendix E

PIC Implementations

This chapter shows the 'C' implementations of Application-I (Thermostat) and Application-II (Multiplier) on PIC12F675.

Code listing 2 Thermostat Implementation on PIC12F675

```
//Test implementation of thermostat application on PIC12F675H
#include <htc.h>
#define _XTAL_FREQ      1000000// oscillator frequency for __delay_us()
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTDIS & INTIO);
unsigned char ones = 0,twos = 0;
void main() {
  int a=0;
  TRISIO = 0;// Configure all IO as outputs
  ANSEL = 1<<0;// make only AN0 analog
  CMCON = 7;// disable comparators (CM = 7)
  OPTION = 0b11000010;// configure Timer0:
  ADCON0 = 0b10000001; // configure ADC
  T0IE = 1;// enable Timer0 interrupt
  ei();    //   and global interrupts
  for (;;) { // Main loop
    a =a+1;//Dummy code to set breakpoint
  }
}
/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
  // *** Service Timer0 interrupt
  T0IF = 0;//Breakpoint set here// clear interrupt flag
  unsigned char temp,thr=50,delta=5;
  // sample analog input
  GODONE = 1;                  // start conversion
  while (GODONE)              // wait until done
    ;
  temp = ADRESL & 0xff;        // get result from ADRESL
  //Statemachine function
  if((temp<(thr-delta)) && (GPIO1 == 1)) {
    GPIO1 = 0;
  } else if((temp>(thr+delta)) && (GPIO1 == 0)) {
    GPIO1 = 1;
  }
  temp=temp+1;//Dummy code to set breakpoint
}
```

Code listing 3 Multiplier Implementation on PIC12F675

```
//Test implementation of 2-Bit Multiplier on PIC12F675H
#include <htc.h>
#define _XTAL_FREQ      1000000// oscillator frequency for __delay_us()
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTDIS & INTIO);
unsigned char ones = 0,twos = 0;
void main() {
  int a=0;
  TRISIO = 0;// Configure all IO as outputs
  ANSEL = 1<<0;// make only AN0 analog
  CMCON = 7;// disable comparators (CM = 7)
  OPTION = 0b11000010;// configure Timer0:
  ADCON0 = 0b10000001; // configure ADC
  T0IE = 1;// enable Timer0 interrupt
  ei();    //   and global interrupts
  for (;;) { // Main loop
    a =a+1;//Dummy code to set breakpoint
  }
}
/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
  // *** Service Timer0 interrupt
  T0IF = 0;//Breakpoint set here// clear interrupt flag
  unsigned char ans;
  // sample analog input
  GODONE = 1;                  // start conversion
  while (GODONE)              // wait until done
    ;
  ones = ADRESL & 0x03;        // get last 2-bits of ADRESL
  GODONE = 1;                  // start conversion
  while (GODONE)              // wait until done
    ;
  twos = ADRESL & 0x03;        // get last 2-bits of ADRESL
  ans = ones * twos;//Multiply the two 2-bit variables
  ans = ans+1;//Dummy code to set breakpoint
}
```