Design of a Power-aware Dataflow Processor Architecture

Ramya Priyadharshini Narayanaswamy

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 20, 2010
Blacksburg, Virginia

Keywords: Power-aware, Dataflow, E-textiles, UPF

Design of a Power-aware Dataflow Processor Architecture

Ramya Priyadharshini Narayanaswamy

(ABSTRACT)

In a sensor monitoring embedded computing environment, the data from a sensor is an event that triggers the execution of an application. A sensor node consists of multiple sensors and a general purpose processor that handles the multiple events by deploying an event-driven software model. The software overheads of the general purpose processors results in energy inefficiency. What is needed is a class of special purpose processing elements which are more energy efficient for the purpose of computation. In the past, special purpose microcontrollers have been designed which are energy efficient for the targeted application space. However, reuse of the same design techniques is not feasible for other application domains. Therefore, this thesis presents a power-aware dataflow processor architecture targeted for the electronic textile computing space. The processor architecture has no instructions, and handles multiple events inherently without deploying software methods. This thesis also shows that the power-aware implementation reduces the overall static power consumption.

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

I dedicate this thesis to my parents, Mr and Mrs Narayanaswamy.

Acknowledgements

I would like to thank my advisor Dr. Mark Jones for his guidance and support all through my graduate life at Virginia Tech. This thesis would not have been possible without him.

I would like to thank Dr. Tom Martin, Co-chair for his ideas and insightful technical discussions throughout the course of this project.

I would like to extend my gratitude to Dr. Paul Plassmann for serving in my committee and for his valuable suggestions in the write-up of this thesis.

I would like to thank my friend and project partner Karthick Lakshmanan who was instrumental in making this project a successful one.

I would also like to extend my thanks to Mr.John Harris, System Administrator for his timely help in setting up the tools.

I would like to thank all my friends who patiently reviewed my thesis and came up with suggestions for improvement.

Finally, I would like to thank my parents Mr. Narayanaswamy and Mrs. Leelavathy, and my brother Mr. Santhosh for their love, support and encouragement all through my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Electronic textiles (called e-textiles) are a class of distributed embedded systems with computing elements directly attached onto fabric, and communication between computing elements is through wires woven into the fabric. The research efforts in the field of electronic textiles broadly fall into two categories.

1. Exploration of application space: Examples include implementation of an e-textile system for applications such as health monitoring [1], location sensing [2], motion capturing, and activity recognition [3].

2. Exploration of design space: A few examples of this category are investigation of a system architecture for e-textiles [4] and a flexible design framework for e-textiles [5].

This thesis explores the design space, and presents an investigation of a processor architecture. The power-aware configurable dataflow processor architecture targeted for event-driven embedded computing space reduces the overall power consumption while exposing the power consumption to the programmer.

1.1 Motivation

This thesis derives motivation from the two-tier system architecture for e-textiles [4]. In the two-tier system architecture, there is a network of Tier 1 nodes and a Tier 2 node which controls this network. Each Tier 1 node consists of a simple microcontroller attached to multiple sensors. A Tier 1 node reads, processes, and converts the data from multiple sensors into a digital format to be sent over the network. A Tier 2 node consists of a more powerful microprocessor that executes the application by collecting sensor values from the Tier 1 nodes. The feasibility of such an architectural approach has been researched, and a proof of concept has been implemented in [6].

In the implementation of the two-tier architecture, a low cost general purpose microcontroller is used for Tier 1 processing. The inputs from sensors are events that control the execution of Tier 1 nodes, and an Instruction Set Architecture (ISA) based processor deploys software methods to handle multiple events. The power consumption associated with such a processor is twofold: first, power is consumed by the control unit that sequences the instruction with the fetch, decode, and execute operations, and second, the addition of software control for processing multiple events consumes considerable power. Such an implementation is not energy efficient for electronic textiles. Energy efficiency is critical in an electronic textile system as it operates autonomously, requiring the battery to last for a long time without recharge.

Therefore, this thesis presents a new class of processor architecture that is suitable for processing at the Tier 1 nodes. Such a computing element is an event-based power-aware configurable dataflow processor architecture with no instructions. The hardware of the architecture would handle multiple events by design. Therefore, the absence of ISA and software layers in this processor architecture would result in a more energy efficient behavior for deployment in Tier 1 nodes. The design of such a processor architecture is explained in Chapter 3.

The timing of events in this dataflow processor architecture is deterministic. This facilitates the power management of the functional units of the architecture, reducing the overall power consumption. The deterministic nature of the power-aware architecture makes it feasible for the programmer to determine the power consumption prior to execution of an application on the architecture hardware.

1.2 Contributions

The work presented in this thesis is part of a complete project that includes the design of functional units and a programming framework for the processor architecture. The contributions of this thesis are:

1. The coarse-grained functional units of the architecture are designed in a configurable and pipelined fashion. Such a design yields a flexible hardware that supports concurrency. Moreover, a template is designed for the functional units of the architecture. This template decreases the design time of any new functional unit by allowing for reuse with minimal changes

2. FIFOs are designed for communication between the high speed network and the low speed architecture instance. The intra-module communication in the architecture, and the communication of the architecture with the FIFOs, are via event buses. The bus is transparent to configuration and data without any arbitration, resulting in a simple communication network

3. The architecture is designed in a power-aware fashion, allowing for significant savings in power consumption. The power-aware operations are transparent to the programmer, however, the power consumption is visible to the programmer

4. A tool automation framework is setup for the implementation of the architecture as an Application Specific Integrated Chip (ASIC). The tool automation framework simplifies

the implementation process, and decreases the manual effort involved in extraction and analysis of results

1.3 Thesis Organization

This thesis is organized as follows: Chapter 2 gives an overview of research efforts relevant to this thesis. Chapter 3 explains in detail the design of the power-aware dataflow processor. Chapter 4 elaborates the implementation of the architecture as an ASIC. Chapter 5 presents the results of this thesis. Chapter 6 summarizes this thesis, and suggestions for future work are also included.

Chapter 2

Background

This section presents an overview of research work related to this thesis. The target environment and the system architecture that help in understanding the motivation for the processor architecture are presented in Section 2.1. Section 2.2 provides an overview of various research related to the design principles of the architecture. Section 2.3 discusses architecture techniques deployed for energy efficiency and low power consumption.

In a distributed embedded computing environment, a number of computing elements coordinate for the successful execution of an application. The execution of an application is controlled by events external to the system. A classic example of such an embedded computing system is the Wireless Sensor Network (WSN). In WSNs, spatially distributed autonomous sensors monitor the physical and environment conditions aiding in home automation [7], industrial process monitoring and control [8], and traffic control [9].

2.1 Related Work in E-textiles

2.1.1 E-textiles

Fabric is an essential part of human life. They are used in many forms including clothes, curtains, sofa covers, carpets, etc. As fabrics are ubiquitous in a civilized society, they serve as a platform for distributed embedded computing for a number of sensing and monitoring applications [10, 3]. Therefore, integrating electronics onto fabric is a form of pervasive computing rightly termed as electronic textiles or e-textiles. In an e-textile system, the computing elements and sensors are attached onto fabric. The communication between spatially distributed sensors is through electric wires woven in the form of threads in the fabric. The wired communication between sensors makes it different from WSN's, influencing the power requirements of the system. E-textiles and WSNs fall under the category of distributed embedded computing, however, they vary in their system architecture, and purpose of computation. Therefore, reuse of techniques and methodologies of WSN for e-textiles is not feasible, and a compelling need for exploration of e-textile computing space exists.

The research efforts in e-textiles include exploration of various design principles and system architecture, and building an e-textile system [11, 6] to evaluate them. This thesis designs a dataflow processor architecture that can be incorporated as a computing element in e-textiles. Thus, it is necessary to understand the system architecture for which this processor is targeted.

2.1.2 Target System Architecture

It is evident from [4] that a flexible and scalable system architecture is achievable with a hierarchy of processors. The hierarchy consists of two levels of processing nodes. The lower level nodes are co-located with sensors, and are many in number. They form the Tier 1 nodes. The upper level consists of one or two processing elements that control all the lower

level nodes. Upper level nodes form the Tier 2 nodes. A typical hierarchy is shown in Figure 2.1.

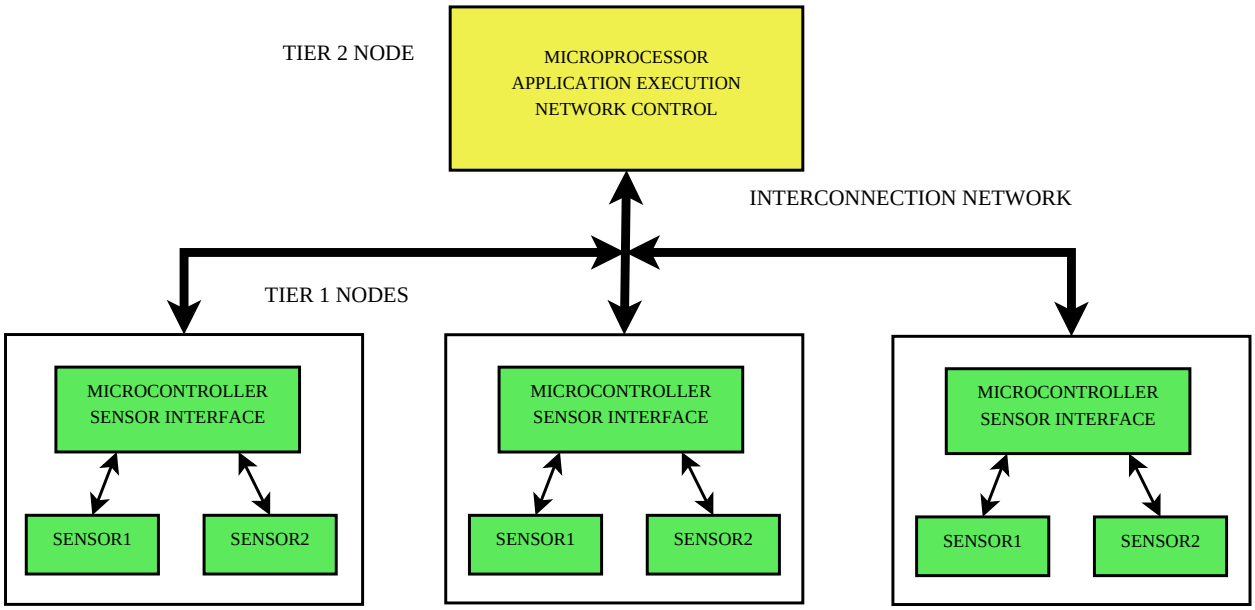

Figure 2.1: Two-tier System Architecture

As shown in Figure 2.1, a Tier 1 node collects data from multiple sensors, and sends them over the network to the Tier 2 node. Tier 1 nodes perform simple operations such as Analog-to-Digial (A/D) conversion, averaging, and filtering of sensor values. These operations are less complex, and a low performance microcontroller is sufficient to handle the complexity of Tier 1 processing. In contrast, the Tier 2 node executes the application, and is responsible for maintaining the network of Tier 1 nodes. This mandates the use of a more powerful microprocessor at the Tier 2 node.

A working model of the two-tier system architecture was implemented in an e-textile garment designed for motion capturing application [6]. A Tier 1 node consists of a low performance Atmega8L micro-controller, and two sensors - a 3-D accelerometer, and a 1-D gyroscope. A single Gumstix Verdex 400xm-bt is used at the Tier 2 node. In the motion capture application, the functionality of Tier 1 nodes is limited to a simple A/D conversion. The sensor outputs serve as events, and multiple events are handled by deploying software methods.

An event driven software model and an optimized software model were designed to handle multiple events. These methods are discussed in detail in the software development chapter of Chong's work [12].

The processor architecture designed in this thesis handles multiple events without deploying software methods, and its performance is targeted to handle processing at the Tier 1 level.

2.2 Related Research Work in Processor Architecture

2.2.1 Dataflow Architecture

The processor architecture designed in this thesis is based upon dataflow model of computation. This section presents an overview of dataflow architectures. Dataflow architecture is a computer architecture in which the execution of instructions is determined by the availability of input arguments. The input to a functional unit contains the address of the instruction to be executed. This is in contrast to the Von Neumann architecture that has a sequencing unit to execute the instructions one-by-one. Dataflow architectures inherently support parallelism, and are deterministic by nature [13]. In static dataflow machines [14], physically separated functional units execute concurrently when all of their inputs are available. Dynamic dataflow architectures [15] are machines that support temporal parallelism. Multiple invocations of the same functional unit are executed concurrently, resulting in a pipelined execution. A dataflow architecture is deterministic as the results of execution are independent of the order of execution of parallely executing functional units. Dataflow architectures are prominent in digital signal processing and image processing systems that require massive parallel computing [16, 17, 18].

WaveScalar [19] is a dynamic dataflow architecture for the execution of general purpose multi-threaded applications. It is targeted for high performance systems, and has an array of processing elements that support dataflow model of execution. In principle, the architecture

designed in this thesis is a variant of dynamic dataflow architecture that supports pipelining, and has an array of functional units. However, the purpose of computation is different from WaveScalar, and is intended for low power embedded systems. Also, the WaveScalar architecture has a hardware abstraction for mapping and scheduling the program instructions to a functional unit. In contrast, the architecture designed in this thesis does not have instructions, and a software framework maps and schedules the applications.

There have been research efforts in the past that tried to exploit the principles of dataflow execution for embedded computing. A transparent dataflow architecture [20] is one such research effort that has a dataflow co-processor in addition to the main control flow based processor. A hardware engine translates the instructions that would be feasible for execution on a dataflow co-processor. Packet Instruction Set Computing architecture [21] is a synchronous dataflow machine targeted for network processing. The architecture is deeply pipelined, and supports networking operations at layers 2-4 of the Open Systems Interconnect model.

All of the dataflow architecture research efforts discussed in this section are ISA-based. Moreover, they have a hardware engine for mapping the instructions onto their computing elements. However, the architecture designed in this thesis has no instructions, and the scheduling of an application is done using software tools.

2.2.2 Event-Driven Architecture

The execution of sensor-monitoring distributed embedded applications is primarily driven by sensor outputs called events. A general purpose microprocessor handles such events by deploying an event-driven software model. For instance, TinyOS [22] is an event driven operating system targeted to WSNs. There have also been research efforts to design a hardware architecture that can handle multiple events. The trend in migration of event-handling from software to hardware is visible in recent processor architectures. Such an effort reduces the software overhead, saving the overall power consumption. In a processor

architecture targeted to WSNs [23], a hardware event queue, and an event processor are parts of the architecture. The responsibility of handling incoming interrupts/events is given to the event processor. A packet processing architecture [24] targeted to wireless base stations consists of an array of functional units called microengines. Every microengine's computation is event-driven with an event queue and an event processing unit.

An event-driven multi-threading architecture [25] targeted to embedded systems handles multiple events by design, and has an inter-network of functional units. The bus network of this multi-threaded architecture is shared by its functional units, and the access of bus is controlled by a bus-master hardware unit. However, the communication bus of the architecture designed in this thesis does not have any hardware arbitration. The functional units of the architecture designed in this thesis are simple hardwired logic based processing units, whereas, the functional units of [25] range from hardwired logic to a powerful ISA-based microprocessor. In this multi-threaded architecture, events and inputs are two different entities, requiring memory blocks to store the inputs until an event arrives at the functional unit. More precisely, it exhibits a process model, where every process needs activation. However, in the processor architecture designed in this thesis, inputs are the events triggering the execution of functional units, and thus, does not exhibit a process model.

The Sensor Network Asynchronous Processor (SNAP) [26] is an event-driven ISA processor targeted to WSNs. Every SNAP contains an incoming and outgoing message buffer for communicating with the network. The SNAP contains a hardware event queue for handling multiple events. Tokens are inserted into the event queue by incoming messages, and are serviced by the main processor. The SNAP architecture is relatively deterministic for execution of an application when compared to other general purpose microprocessors as there is no cache or virtual memory leading to nondeterministic memory latencies. However, there is a DRAM for storing and executing instructions. A design principle of the architecture designed in this thesis is similar to SNAP, with FIFOs for communication with the network. In contrast to SNAP, the architecture designed in this thesis is primarily a non-ISA.

2.2.3 Configurable Architecture

Configurable computing machines achieve the performance of executing an application on hardware, and the flexibility of software through reconfiguration. Partial reconfiguration is a mechanism by which only a portion of the chip is reconfigured. There are applications for which complete reconfiguration of an area of the chip is not required. For instance, changing the co-efficients of a filter or changing a constant used in computation. In such scenarios, coarse-grained or device level reconfiguration is useful. Wormhole routing based reconfiguration [27] proposed an architecture that deploys device level reconfiguration. The devices of the architecture have unique addresses which are used by stream-based packets for configuration. Stream-based packets contain both configuration and data for the execution of application. The devices reconfigure on reception of a packet, and execute using the data obtained from the same packet. Such a scheme reduces the interconnection complexity as both configuration and data use the same signal path. The architecture designed in this thesis is based on the coarse-grained configuration methodology of configurable computing machines, and the signal path is transparent to configuration and data. However, configuration and data are not embedded into one single packet. The configuration packets configure the devices, and the data packets trigger the execution of devices.

The Elemental Computing Architecture (ECA) [28] is another class of configurable machines that deploys device level configuration. The devices of ECA are non-homogeneous and highly pipelined. The devices are clustered, and clusters use FIFOs for communication. A hierarchy of clusters aids in the execution of complex applications. This architecture supports execution of multiple tasks on its hardware by deploying a hardware operating system. This hardware operating system maps the tasks onto the devices, and reconfigures the devices. The proposed architecture employs similar device level configuration and FIFOs as ECA. In contrast to [28], the architecture designed in this thesis does not have a hardware operating system. Software tools are used to map the application tasks onto the functional units of the architecture.

Both [28] and [27] are capable of run-time reconfiguration. In contrast, in the architecture designed in this thesis configuration completes prior to application execution. Moreover, the devices of the architecture designed in this thesis are not truly reconfigurable. For instance, a timer device is always the same, and it cannot be completely reconfigured to function as an A/D converter. However, the value of the counter associated with a timer unit is configurable.

2.3 Architecture Techniques for Energy Efficiency and Low Power Consumption

The life-time of a sensor monitoring system depends on the energy efficiency of the sensor nodes. Many such systems are battery powered, requiring the sensing, computation, and communication of sensor nodes to be optimized for energy efficiency. It is evident from [6] that Tier 1 nodes have low event rates, and they operate at a low frequency. Therefore, there is a need for an architecture that matches the low computational demand of Tier 1 nodes. This section presents an overview of research efforts in energy efficient architectures for low event rate sensor monitoring system.

The operating voltage of a processor is higher than the threshold voltage of the CMOS transistors of the processor. The performance of a processor even with the minimal operating voltage appears to be higher than needed for low performance sensor network nodes [29]. Therefore, an architecture that operates at a sub-threshold voltage was proposed [30]. The sub-threshold circuits are not ordinarily robust across variations in process and temperature, requiring research efforts to design processors with robust behavior.

An event-driven architecture for sensor applications with device level power management was proposed in [31]. The architecture consists of tailor-made functional modules connected via a bus. The architecture includes an event processor to control the execution of an application,

and a general purpose microcontroller to support any operations not handled by functional modules. The resource usage of this architecture is event-driven, facilitating switching off any unused modules to reduce the overall power consumption.

An ultra low power processor architecture was proposed for cubic millimeter sized sensor nodes [32]. The processing units are power gated with high threshold transistors, reducing the power consumption in stand-by mode. The power gating approach in combination with low voltage ROM, low leakage memory cells, and a compact ISA yields an energy efficient architecture for the battery powered sensor nodes.

All of the processor architectures discussed in this section share common principles. They do not have any software control for handling events, reducing the overall power consumption. Moreover, the processor architectures are tailor-made for specific targets such as WSNs with low event rates. Such a design is more energy efficient for the purpose of computation rather than deploying general purpose microprocessors. A similar principle is employed in the architecture designed in this thesis. Power gating approach is employed for the power management of functional units as in [32], and the power management principle is at the device level similar to [31]. However, all of the special purpose architectures discussed are ISA-based. The instruction fetch, decode, and execute operations of the control unit consumes considerable power, and a compact ISA design was required for energy efficiency [30, 32].

The current trend in CMOS process technology plays a significant role in processor architecture design. A more dense and a higher performance processor is feasible with continuing decrease in feature size. This improvement in performance is marred by the dominance of leakage power over dynamic power in deep sub-micron technologies [33]. Leakage-aware design techniques vary with process technology. For instance, adaptive back body biasing is a fine-grained leakage power control mechanism effective at 130nm process [34]. The deployment of the technique requires additional circuitry with a power penalty. However, the same technique is not effective for processes below 100nm. As the performance demands of sensor nodes are low, [34] suggests using an older process technology for the architectures

targeted to sensor nodes, thereby reducing the leakage power consumption. There is less emphasis on the feature size and the overall size of the chip. However, in e-textiles computing space, the computing element is attached onto the fabric, and a smaller chip that is non-invasive to users and environment is required. Therefore, the processor architecture researched in this thesis is designed with leakage-aware principles suited for deep sub-micron process technology of 90nm and below.

The power savings obtained from asynchronous circuit design techniques does not scale well with the improvements in process technology as considerable amount of power is dissipated in orchestrating the handshake process. To reduce this power consumption, a technique was proposed in [35]. Multiple voltage islands are created enabling different parts of a single chip to function at multiple voltage levels. Level shifters or voltage translators are used between signals spanning multiple voltage domains. A similar voltage island phenomenon is utilized for reducing the power consumption of the architecture designed in this thesis. In contrast to operating the voltage islands at various voltage levels, the voltage islands of the architecture designed in this thesis are switched off when not in use.

Chapter 3

Design of Processor Architecture

This chapter presents in detail the design of the power-aware dataflow processor architecture targeted for low event rate distributed embedded computing. Section 3.1 presents an overview of features of the architecture. Section 3.2 elucidates the design of functional and communication units of the architecture. Section 3.3 elaborates the leakage-aware design deployed for reducing static power consumption.

3.1 Overview of the Processor Architecture

The processor designed in this thesis is a dataflow processor with no instruction set. The processor architecture does not deploy a software process model for handling events like the general purpose computers. The advantage of the absence of a process model for executing an event-driven application is that there are no multiple logic tasks that execute on a single processing element leading to run-time resource conflicts. Therefore, the architecture designed in this thesis does not suffer from run-time resource conflicts, giving predictable performance. The deterministic nature of the architecture facilitates accurate computation of a minimum clock rate for executing an application.

The building blocks of the architecture are independently operating functional units. The functional units are designed in a configurable and pipelined fashion, and their execution is event-driven. The functional units communicate with one another through event buses. The processor architecture is targeted for distributed embedded computing, and therefore, is designed with a FIFO for communicating with the network. The FIFO synchronizes the flow of events between the low speed processor and the high speed network. There is no central control unit for sequencing the execution of functional units, and each functional unit executes autonomously if it receives the required input. The flow of data (also called events) between the functional units facilitates the execution of an application on the architecture.

The processor architecture may be deployed in a variety of networks, therefore, a network interface is required for translation of data between the processor and the network. The aim of this thesis is to explore the design principles of the architecture by implementing functional units, event buses, and event FIFOs. The validation of the architecture is possible without any network interface, and therefore, this thesis does not explore the design space of network interfaces.

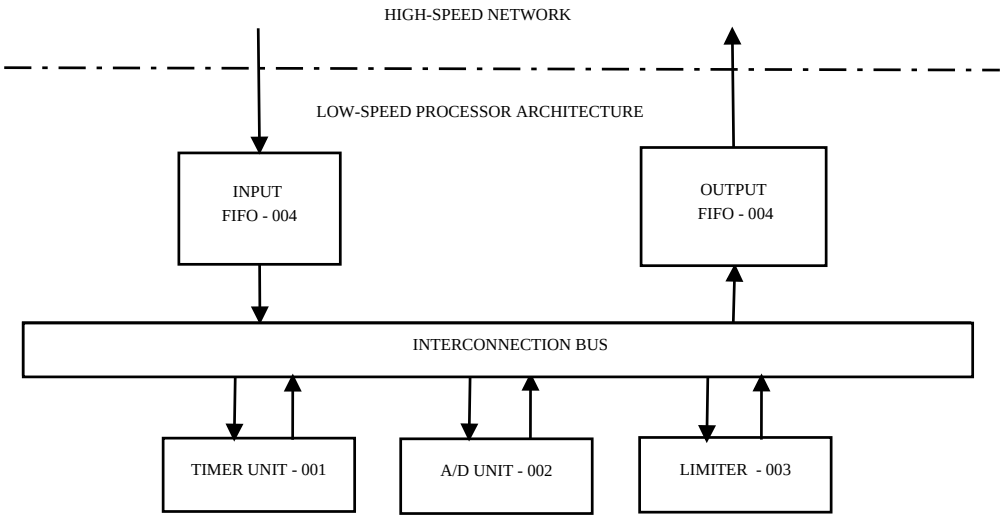

Figure 3.1: Sample Architecture Instance

A sample architecture instance is shown in Figure 3.1. The architecture consists of a timer unit, an Analog to Digital (A/D) converter unit, and a limiter unit. The timer unit gen-

erates events periodically. It is a source unit, and does not require inputs to trigger the execution. Upon reception of an input event, the A/D unit samples its analog channel, and begins the conversion. Similarly, upon reception of an input event, the limiter unit starts its execution. The functional units are connected via an event bus, and FIFOs are present for communicating with the network. The network configures the architecture instance through the input FIFO, and the results are sent to the network through the output FIFO. All of the functional units have unique addresses, and the FIFOs have address of the network.

As there are no instructions to program the architecture, applications are represented as high-level descriptions, and software tools transform these high-level descriptions onto configuration events. A sample application is shown in Figure 3.2. The application intends to sample an analog channel once in 10 ms, and the digitized value is limited. There are 3 nodes in the application that aid in achieving the functionality of the application. The timer node generates periodic timer events destined to the A/D converter. The digitized value from the A/D converter are destined to the limiter node. The parameters of the application nodes are shown in Figure 3.2. These parameters are also used for generating the configuration events of the functional units.

The functional units operate autonomously and continuously once configured by setting its module active configuration bit. The destination address configuration contains the address of a functional unit to which the data events are intended. The delay configuration specifies the number of cycles that a functional unit needs to wait before transmitting events onto the event bus, and the software tools generate these delay values such that there is no collision of events on the bus. Therefore, the design of an event bus in hardware is simple without any arbitration. Other internal configuration required for the functional units are also generated by software tools. The event bus carries configuration events for the functional units prior to execution of an application, and data events during the execution of an application.

The operation of the processor is explained by the execution of a sample application on the sample architecture. The architecture is configured to periodically sample an analog sensor

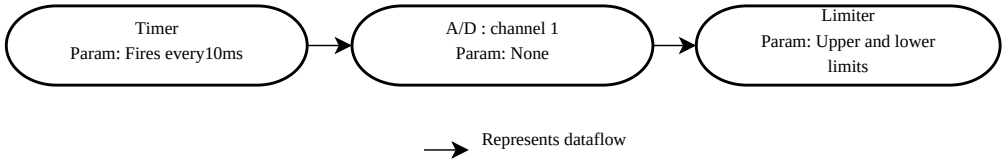

Figure 3.2: Sample Application Graph

value, and to check if the digitized value is within a predefined range using the limiter. The timer generates periodic timer events (with A/D converter as destination address). Upon reception of a timer event, the A/D converter samples the analog channel and starts conversion. The digital value from the A/D converter is an event addressed to the limiter unit. The limiter unit checks if the value is between predefined upper and lower limits before transmitting it to the network through the output FIFO. The lower or the upper limit is sent to the network for any digital value that lies outside the configured range.

The functional units of the processor architecture become operational when configured by setting its module active bit. The timer starts off once it is configured, and may generate an event before other functional units are configured by the network. An output event from the timer might be ignored by the A/D converter if it is not configured. To avoid such a scenario, the functional units are always configured in the reverse order of dataflow of the executing application. The configurations for execution of the above discussed application are given in Table 3.1. A sample timing diagram depicting the execution of functional units, and transfer of events on the bus is presented in Figure 3.3. The limiter unit is configured first, and the timer unit is configured at the end of configuration phase. The timer generates an event at cycle 180, and this repeats periodically. The A/D converter takes 10 cycles for execution, and generates an output event at cycle 195. The digitized value from the A/D converter triggers the execution of the limiter which takes 5 cycles for execution, and transfers the data onto the network. The timing diagram shows that once the application is configured, the application executes periodically on the architecture instance.

Table 3.1: Sample Configuration

Module	Destination address	Delay value	Internal configuration	Module active
Timer	2	0	34	1
A/D	3	0	–	1
Limiter	4	0	(3,10)	1

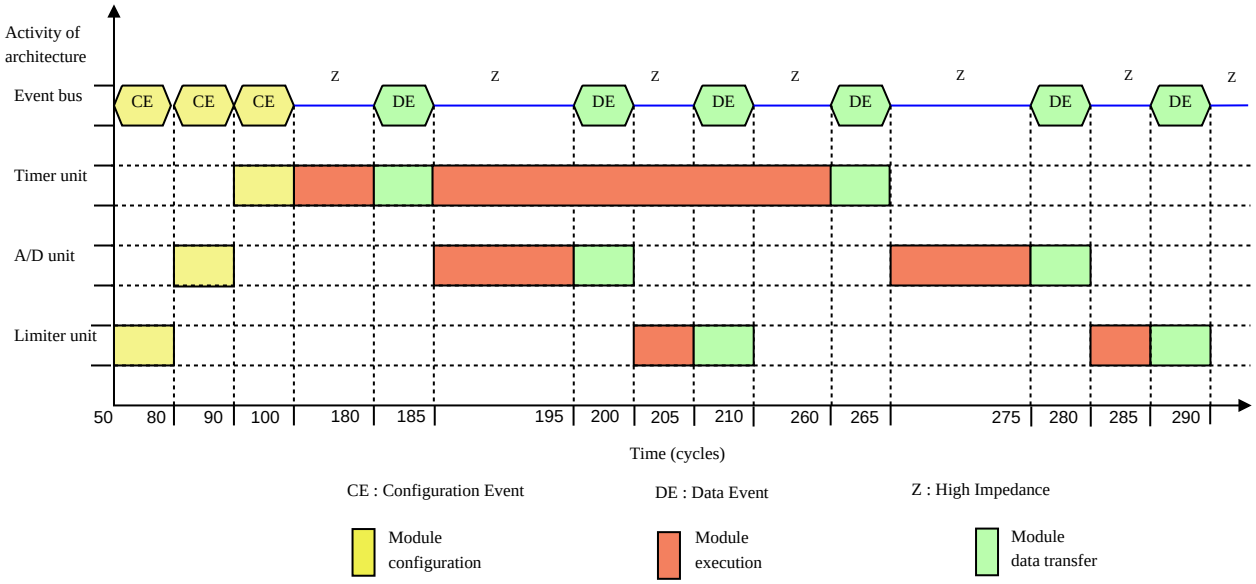

Figure 3.3: Sample Timing Diagram

3.2 Design of Processor Architecture

This section describes the design of functional and communication units of the processor architecture. The architecture and its functional units are designed in a parameterizable fashion. The architecture is built around transfer of configuration and data events that aid in the execution of an application. The transfer of events is through fixed length packets. The packet structure is an important factor in the design of functional and communication units. This version of the architecture is designed to handle a packet length of 12 bits for both configuration and data. The packet structure for transmission of configuration and data events are shown in Tables 3.2 and 3.3.

Table 3.2: Packet Structure for Configuration

A_2	A_1	A_0	C/D'	WR/INT'	CA_1	CA_0	CD_4	CD_3	CD_2	CD_1	CD_0

Table 3.3: Packet Structure for Data

A_2	A_1	A_0	C/D'	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

The capabilities and limitations of the processor architecture are drawn from this packet structure. The packet structure is explained below.

$A_{2..0}$

The most significant bits of the packet represent the address of the destination functional unit for the event.

C/D'

Packet identifier bit. '1' denotes a configuration packet, and '0' denotes a data packet

WR/INT'

Configuration identifier bit. '1' indicates configuration in the wrapper. '0' indicates internal configuration specific to a functional unit

$CA_{1..0}$

Bits representing the address of configuration registers within a functional unit. As the internal configuration is specific for a functional unit, the boundary between address and configuration bits is not stringent

$CD_{4..0}$

Bits representing the configuration in a configuration event

$D_{7..0}$

Bits representing the data in a data event

3.2.1 Design of Communication Units

This section presents the design of communication units of the processor architecture. The subsequent subsections elaborate the design of the event bus and the FIFOs of the architecture.

3.2.1.1 Event Bus Design

The event bus consists of data lines and a single control line called bus_busy_signal. The data lines carry configuration and data events, and are in high impedance state when not in use. The control line signals the start of transfer of events on the bus, indicating that the bus is not in high impedance state. The input bus-interfaces of functional units start capturing the data in the bus when the bus_busy_signal is at logic 1.

The number of data lines determines the width of the event bus, and multiple architecture instances are feasible by varying the bus width. The bus width determines the number of cycles required for transmission of events. A bus with a width of 4 takes 3 cycles for complete transfer of events, whereas, a bus with a width of 12 takes one cycle for event transmission. The transmission is such that the most significant bits are transferred onto the bus before the least significant bits. The design of the bus is such that there is no collision detection or avoidance logic associated with it, and there is no logic to differentiate between configuration and data events. Thus, the bus is transparent for the transfer of configuration and data without any arbitration. The transmission of events on bus for various architecture instances is shown in Figure 3.4. From Figure 3.4, it is evident that the control line of the bus is ON for one cycle indicating the start of transmission, and OFF for the remaining cycles. The bus is in high-impedance state when there are no events for transmission.

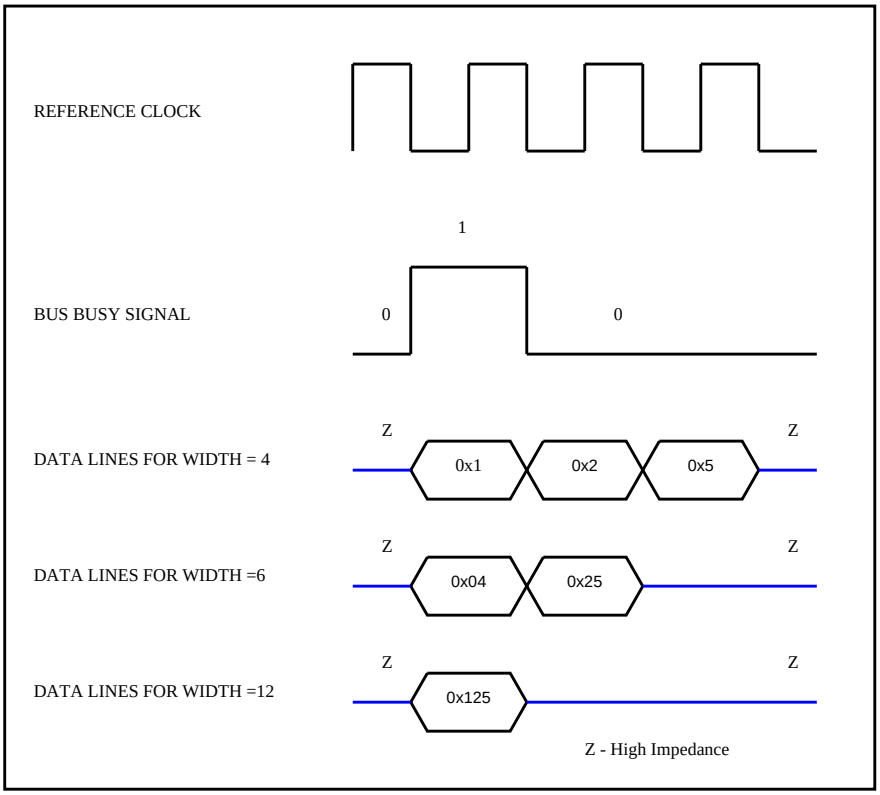

Figure 3.4: Transmission of Events for Various Bus Widths

3.2.1.2 Event FIFOs Design

FIFOs are designed to enable communication between the low speed processor architecture and the high speed network. The FIFOs are designed in a parameterizable fashion controlling their depth. The depth of FIFO depends on the speed of the network, the speed of the processor, and the distribution of incoming traffic from the network. Exploration of these factors to determine the depth is beyond the scope of this thesis. However, the parameterized approach provides scope for such a design space exploration.

The input FIFO is used for transfer of configuration and data events from the network to the processor. During transfer of data events, the FIFO must ensure that the event bus is free, and no other functional units are transmitting data events. Thus, the FIFO has to ensure that there is no collision of events on the bus. The deterministic nature of the architecture facilitates in knowing the schedule of events on the bus apriori. The cycles in which the

event bus is available could be present as configuration in FIFO, and the FIFO transmits data events only when the event bus is free. There are two ways in which this information could be represented. The schedule of the event bus could be represented as a form of look-up table in configuration registers of the input FIFO. The second approach is based upon the busy cycles and the free cycles of a schedule period. During execution of an application, every period of a schedule has a number of busy cycles (during which the functional units are active, transmitting data events onto bus), followed by a number of free cycles (during which there are no events on bus from the functional units). This is shown in Figure 3.5. The design of the input FIFO deploys the second approach. The second approach limits the input FIFO to access the bus only in the free cycles of a schedule. This approach, however, simplifies the amount of configuration information. Instead of storing the complete schedule of the event bus in a look-up table, only the number of busy cycles and the number of free cycles are stored.

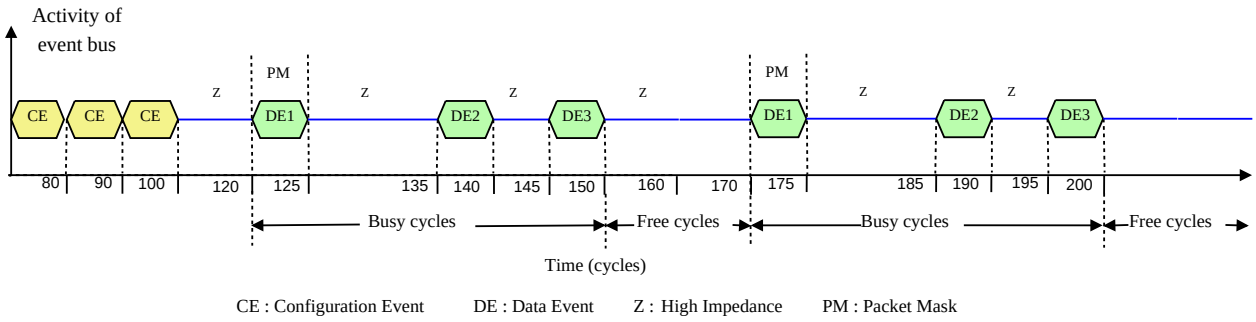

Figure 3.5: Activity of Event Bus

Therefore, configurations associated with the input FIFO are module-active, number of busy cycles, number of free cycles, and packet-mask. The start of a busy cycle is indicated by the transfer of first data event on the bus in every schedule period, and this information is present as packet-mask configuration in the input FIFO. The packet mask is 4 bits: the first 3 bits indicate the destination address of the first data packet, and the 4^{th} bit is a data packet identifier. The input bus-interface looks for such a pattern in the event bus. Once such a pattern is matched with the data packet (match would happen only at the first data packet) on the event bus, the busy cycle counter is started. On completion of busy cycle

counter, a free cycle counter is started. The input FIFO transfers data onto the event bus during these free cycles. All of the data events present in the input FIFO, and any additional events entering from the network during the free cycles are transferred onto the event bus. The number of data events passed onto the event bus depends on the number of free cycles, and the bus width of the architecture. As long as free cycles are available, the input FIFO could transfer data onto the event bus. There is no upper limit on the number of data events that could be transfered onto the event bus from the input FIFO.

Therefore, bus-interface of FIFO looks for first data event based on the packet-mask configuration, keeps track of the number of busy cycles, and transfers data onto the event bus in free cycles, avoiding collision of events on the bus. However, initial configuration events from the network are transferred onto the event bus without any wait cycles. This two-way operation is achieved by the module-active configuration in which the input FIFO looks for free cycles for transmission. The block diagram of an input FIFO is shown in Figure 3.6. The busy and free cycles in a schedule period are shown in Figure 3.5. The busy and the free cycles of the input FIFO are obtained from the time period of the timer. This is equivalent to embedding the global clock information in different units of the architecture. This is because of the design approach chosen for the input FIFO.

The depth of the input FIFO is fixed for an architecture instance. The network sends the configuration as bursts of events [36]. The number of events in the burst is determined by the speed of the network, the speed of the processor architecture, and the depth of the FIFO. After this burst transfer, the network waits until the input FIFO is empty, and then it sends the next burst. The FIFO becomes empty after it transmits all of the events in the burst onto the event bus. Such a burst-based scheme ensures that the input FIFO is not swamped by excessive flow of data from the high-speed network. The same principle is applicable if there are many data packets (greater than the depth of the FIFO) to be transferred from the network to the processor during free cycles. The data events from one burst could be transmitted onto the event bus in multiple free cycles, if they could not be accommodated in a single free cycle.

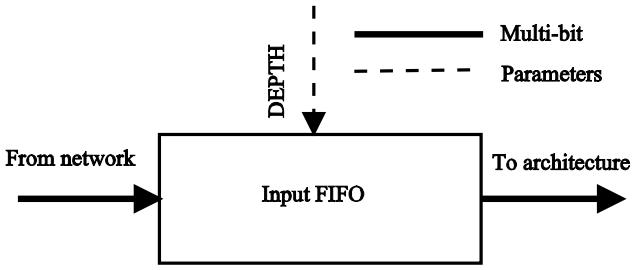

Figure 3.6: Input FIFO

The output FIFO is addressable by functional units of the architecture for transmitting the results of execution onto the network. The bus-interface associated with the output FIFO collects data from the event bus, and forms a 12-bit packet to send to the network. The output FIFO has no configuration registers controlling its operation. The block diagram of output FIFO is shown in Figure 3.7.

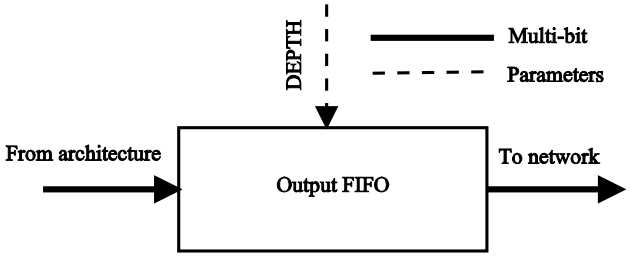

Figure 3.7: Output FIFO

3.2.2 Design of Functional Units

This section elaborates the design of functional units of the architecture. In component-based design [37], every functional unit has a common wrapper module for interaction with the interconnection network, and an internal module for executing the intended functionality. The interface between the internal module and the wrapper is fixed, and changes in interconnection network do not affect the design of the internal module. Such a flexible approach reduces the design effort of a functional unit because the internal module only needs to be plugged into the template. The block diagram shown in Figure 3.8 depicts the modules in a functional unit.

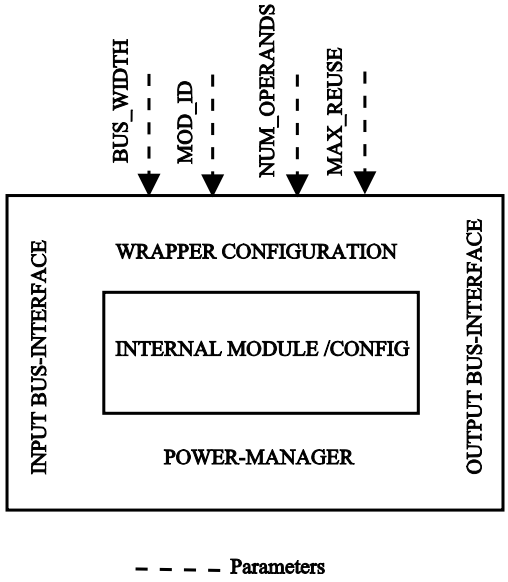

Figure 3.8: Template of a Functional Unit

The template is designed in a parameterizable fashion facilitating flexibility. MOD_ID is the hardware address of a functional unit. BUS_WIDTH parameter facilitates flexibility in the design of bus-interfaces for variations in width of the event bus. NUM_OPERANDS specifies the number of inputs required for execution of a functional unit, and ensures the start of execution only after all the required inputs are received. For instance, an A/D converter requires one input to start the conversion, and its NUM_OPERANDS parameter is set to 1. A subtracter requires two inputs to start its execution, and its NUM_OPERANDS parameter is set to 2. Every functional unit is designed in a pipelined fashion enabling reuse of hardware units for multiple nodes in an application. MAX_REUSE is the maximum limit on the number of times that a functional module would be reused in a single schedule.

Input bus-interface, Output bus-interface, Wrapper configuration, and Power-manager modules form the wrapper of a functional unit. The wrapper modules are parameterizable, facilitating reuse. The internal module comprises the functionality and the internal configuration associated with a specific functional unit. The modules of a functional unit take multiple cycles for execution, and their execution is dependent on the interconnection network and the functionality of the internal module. Therefore, handshake signals are employed for

signaling the completion of execution of a module, and the use of results by its adjacent module. Moreover, the handshake signals aid in the pipelined execution of a functional unit.

The block diagram for every module of a functional unit follows this notation: The signals at left of the block are inputs, the signals at right of the block are outputs, the signals at bottom of the block are handshake signals, and the signals at top of the block are parameters.

3.2.2.1 Input Bus-interface

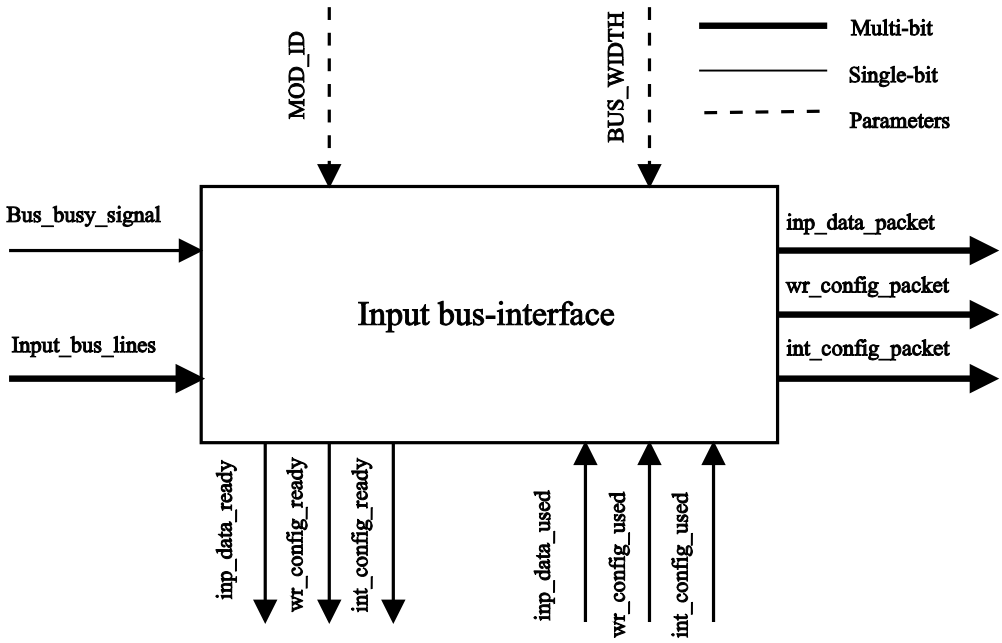

Figure 3.9: Input Bus-interface

The input bus-interface module forms a 12-bit event packet from the bus, and is shown in Figure 3.9. The number of cycles required to form an event packet is dependent on BUS_WIDTH. An MSB first transmission ensures that the destination address of the event is transferred onto the bus before data. All of the functional units connected to event bus read in the address bits, and only the functional unit whose MOD_ID matches the address bits of the transmitted event forms the packet. All other functional units reject the event. The input bus-interface analyzes whether the received packet is either configuration or data,

and the respective sub-packet for configuration or data is formed.

3.2.2.2 Wrapper Configuration

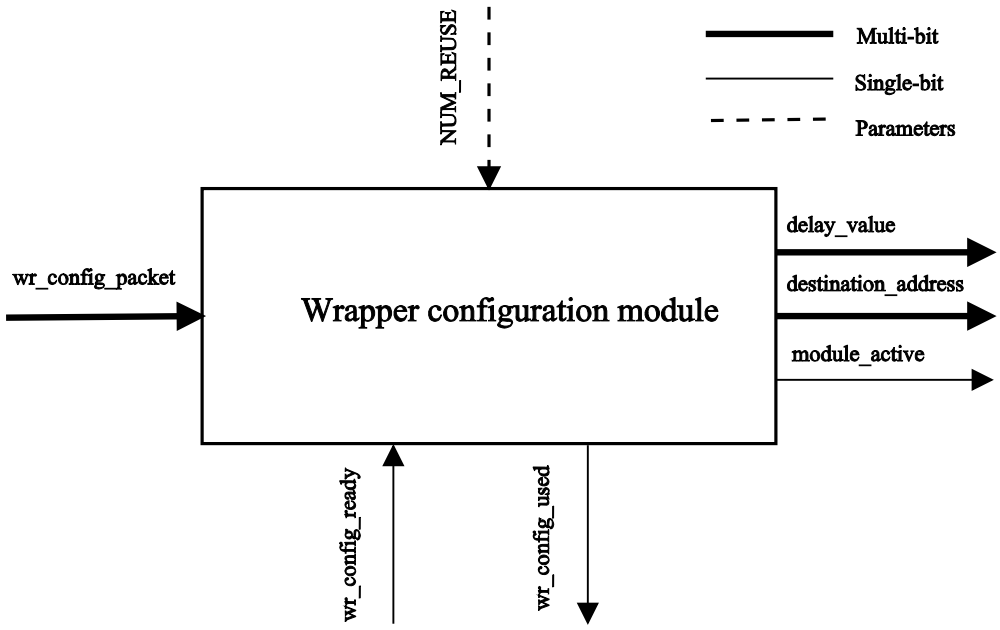

Figure 3.10: Wrapper Configuration Module

The wrapper configuration module is depicted in Figure 3.10. This module configures the registers in the wrapper that aid in formation and transmission of event packets without collision onto the bus. Upon reception of a configuration packet from the input bus-interface, it sets the configuration registers, and asserts the wr_config_packet_used signal to reset the wr_config_packet_ready signal of the input bus-interface. The address of registers and the configurations associated with them are listed in Table 3.4.

The module active indicates if a particular functional unit is operational. The num_used configuration indicates the number of times a hardware functional unit is reused by an application, and is less than or equal to MAX_REUSE of a functional unit. The destination address indicates the address of the functional unit to which a generated data event is intended. The delay value indicates the number of cycles that the output bus-interface has to wait before transferring an event onto the bus. The destination and delay registers are

Table 3.4: Wrapper Configuration

Address	Configuration register	# of bits
00	module active	1
01	num_used	5
10	destination/delay for output_1	3/5
11	destination/delay for output_2	3/5

configured by two packets addressed to the same register in order. Moreover, if a functional unit is reused, every invocation would require different destination and delay values. There are, therefore, a set of destination and delay registers associated with every output.

3.2.2.3 Internal Configuration

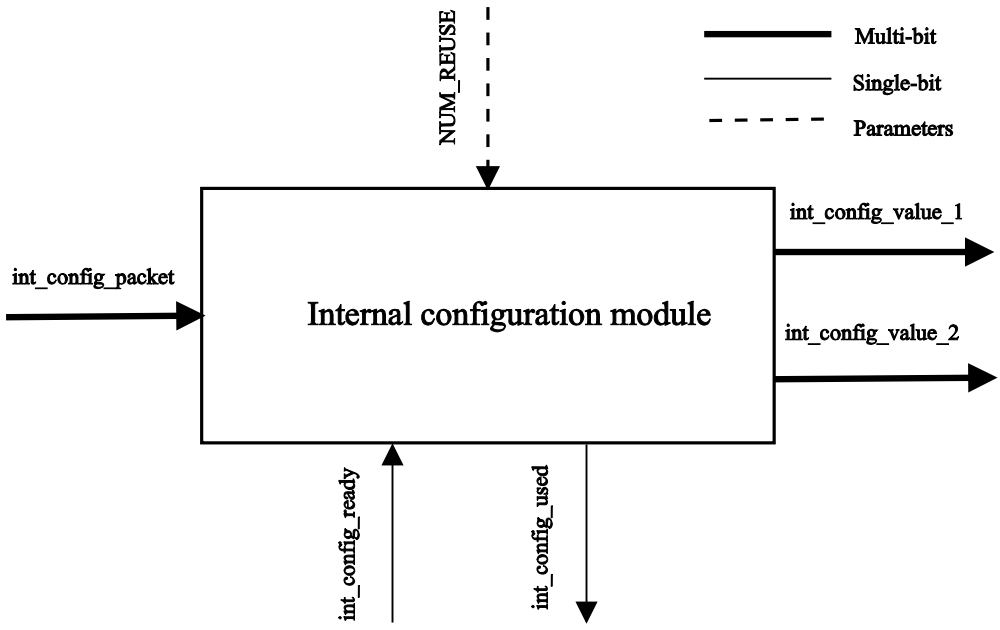

Figure 3.11: Internal Configuration Module

This module receives the internal configuration packet from the input bus-interface, sets the internal configuration, and asserts the packet_used signal. The internal configuration is specific to each type of functional unit. For instance, the internal configuration of a limiter

Table 3.5: Internal Configuration

Address	Configuration register	# of bits
0	upper limit register	6
1	lower limit register	6

unit consists of an upper limit register and a lower limit register with the address space and values as denoted in Table 3.5. The block diagram depicting an internal configuration module is shown in Figure 3.11.

3.2.2.4 Internal Module

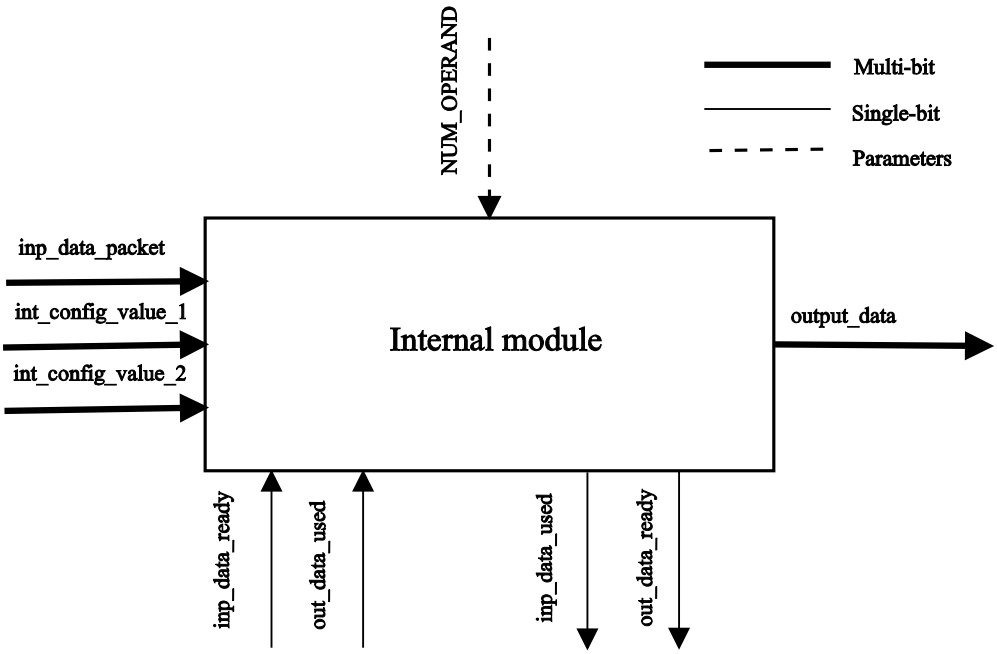

Figure 3.12: Internal Module

The internal module implements the functionality of a hardware unit. The block diagram is depicted in Figure 3.12. The internal module receives input from the input bus-interface, executes the functionality with the aid of internal configuration, if any, and generates the result. The internal module executes only if it receives all the inputs required for execution,

and if the results of previous execution have been transferred to the output bus-interface. The internal module asserts packet_used signal to reset the packet_ready signal from the input bus-interface. The results are transferred onto the output bus-interface along with a data_ready signal indicating the availability of results. For instance, the inputs for execution of a subtracter unit are collected by its input bus-interface. Upon reception of all the inputs from the input bus-interface, the internal module does subtraction, and transfers the result to the output bus-interface.

3.2.2.5 Output Bus-interface

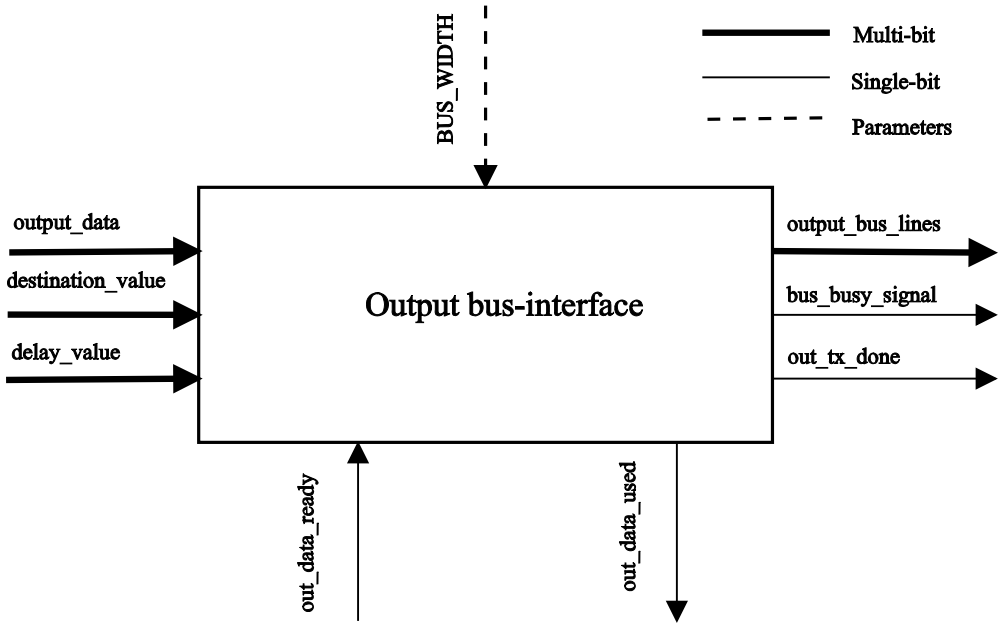

Figure 3.13: Output Bus-interface

The block diagram of the output bus-interface is shown in Figure 3.13. The data packet is formed by combining the bits of destination address from the wrapper configuration, and the results from the internal module. The data packet is transferred onto the event bus after the specified number of wait cycles given by delay configuration.

3.3 Leakage-aware Design of Functional Units

This section describes the power-aware design of the processor for reducing static power consumption. A leakage-aware design is achieved by switching off the power to modules of a functional unit during execution. The configuration registers are not powered-down because they store the destination and delay values. If they are powered-down, the values have to be configured after power-up, and such a reconfiguration increases the clock cycles required for a functional unit execution.

The internal and the output bus-interface modules are idle most of the time, and are used only when the data required for a functional unit execution arrives at the input bus-interface. Therefore, these modules could be powered-down when not in use to reduce static power consumption. The power-up and power-down functionality is achieved by grouping modules into power domains, and controlling power to these power domains by a switch [38]. Thus, a coarse-grained power gating approach is utilized for reducing static power consumption of a functional unit. The design deploys an isolation block enclosing every power domain to prevent the invalid signal transmission [38] during the power-down state. The power-manager module in the wrapper of a functional unit generates the control signals to switch ON/OFF the power gating switch, and to activate the isolation blocks prior to power-down. The isolation control is deactivated after the module is powered-on. The power manager blocks for the internal and the output bus-interface modules are shown in Figures 3.14 and 3.15.

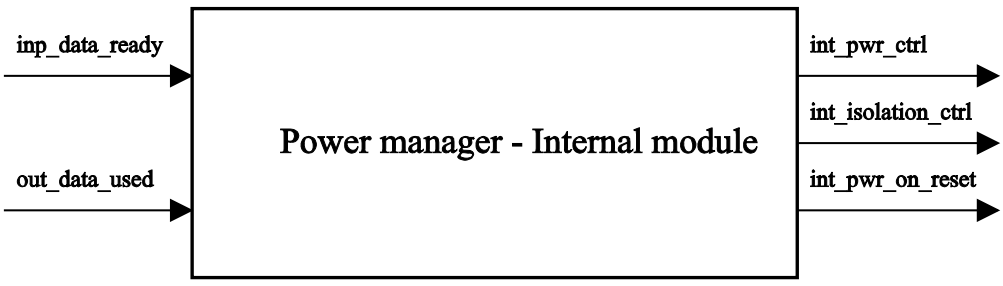

Figure 3.14: Power Manager for the Internal Module

The internal and the output bus-interface modules are powered-down during the configu-

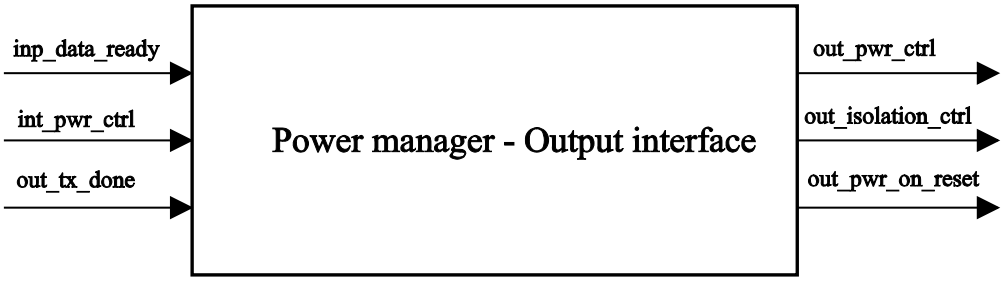

Figure 3.15: Power Manager for the Output Bus-interface

ration phase. During application execution, these modules are powered-up when the input bus-interface receives all the inputs required for the functional unit execution. The internal module is powered-down when the results are transferred to the output bus-interface, and no more inputs are available. Similarly, the output bus-interface is powered-down when the data packet is transferred onto the bus, and there are no more results from the internal module. The completion of execution of the input bus-interface module triggers the power-up of the internal and the output bus-interface modules. The completion of execution of the internal module triggers its power-down. The functional unit deploys handshake signals to indicate the completion of execution of its modules. These handshake signals are used by the power manager to control the power gating switch and the isolation circuitry.

It is not feasible to switch off the input bus-interface during the busy cycles of a schedule, as it filters the packets from the bus. However, as mentioned in the input FIFO section, every schedule period has free cycles during which there is no activity on the bus, and it is possible to switch off the input bus-interface during free cycles. Similar power saving techniques are employed for the design of FIFOs. The input FIFO is powered-down after the configuration phase, and it is powered-up when there are events from the network to the processor architecture during execution. The output FIFO is always in a powered-down state, and it is powered-up when there are events from the processor architecture to the network.

From the above discussions, the modules that are required to be ON during the configuration and execution phases are summarized. During the configuration phase, the input FIFO, the

input bus-interfaces and configuration registers of all of the functional units are ON. The output FIFO, the internal and the output bus-interface modules of all of the functional units are OFF. During busy cycles of the application execution, the internal and the output bus-interfaces of the functional units are turned ON when required. The output FIFO is turned ON in the busy cycle only when there is an event from the processor to the network. The input FIFO is OFF it is empty, and if there are no events from the network. The configuration registers are always ON in the application execution phase. During the free cycle, the internal, and the output bus-interface modules are OFF. The output FIFO is also OFF. If the input FIFO is not empty, the input FIFO and the input bus-interfaces are not turned OFF. Once the input FIFO is empty, the input FIFO and all of the input bus-interfaces are turned OFF. Additionally, if there an event from the network in the free cycle, and if the input FIFO and the input bus-interfaces are OFF, then they are turned ON. Moreover, the input bus-interfaces are turned ON at the end of the free cycle.

The power-aware design discussed in this section does not require any control by the programmer of the application. These power-aware operations are completely automatic, and they only depend on the handshake signals generated by the modules of the functional unit. Thus, the power-aware operations are transparent to the programmer.

The overheads associated with a power-aware design are an increase in execution time required for operation of a functional unit, and a slight increase in area and power due to the inclusion of power management circuits. The power-up sequence takes two cycles, including one cycle for power-on reset. This increases the total execution cycles of a functional unit by 2. The power savings obtained from such a design are analyzed by executing sample applications on the processor, and the results are summarized in Chapter 5.

Chapter 4

Implementation Framework

The design principles and features of the processor discussed in Chapter 3 are evaluated by implementing the architecture as an ASIC, and this chapter describes the ASIC implementation. Section 4.1 describes the Hardware Description Language (HDL) used for representing the architecture. Section 4.2 elaborates the ASIC implementation using Synopsys tools [39]. Section 4.3 elucidates the leakage-aware implementation of the architecture using the Unified Power Format (UPF) [40].

The architecture is implemented in a hierarchical top-down fashion with the top-level file consisting of instances of functional and communication units. Again, a functional unit is implemented as a hierarchy of modules forming the wrapper and the internal modules. A template for the functional unit is implemented, and the same template is reused across multiple functional units. The top-level of sample functional units is given in Appendix A. It is evident that a functional module is formed by stitching together the wrapper and internal modules, and the wrapper structure is the same for different functional units.

4.1 Choice of HDL - SystemVerilog

The specifications of the architecture are derived from the design principles, and are transformed into logic expressions. This logic design phase involves design of finite state machines, combinational, and sequential digital systems for representing the architecture. The functionality of various modules is represented by the transfer of data between registers, i.e., Register Transfer Level (RTL) based logic design is deployed. This thesis uses SystemVerilog [41] for implementing the RTL design for the reasons described below.

SystemVerilog constructs facilitate the use of data types and multi-dimensional arrays in the ports of a module definition. This allows for concise expression of the functionality of modules, reducing the number of lines in HDL [42]. SystemVerilog provides a high-level abstraction called an Interface for inter-module communication. An interface is a collection of signals, wires, logic gates, and functions bundled into one unit. This feature of SystemVerilog is advantageous for implementation of event bus of the architecture. The data lines of the bus are bundled together to form the bus interface, and this bus interface is instantiated in the ports of functional units of the architecture. Any changes to the number of data lines of the event bus are localized in the Interface, and this propagates to all the functional units that instantiate the Interface [42]. The use of SystemVerilog determines the tool-chain for implementing the architecture. An electronic design automation tool-set that supports synthesizable constructs of SystemVerilog is required. Therefore, the Synopsys tool-chain is used for logic synthesis, physical synthesis, verification, and power analysis.

4.2 Tool Automation Framework

This section describes the tool-chain, and the automatic tool framework for implementation and verification of the architecture as ASIC. The tool automation framework is shown in Figure 4.1, and is integrated with the programming framework [43]. The programming

framework encompasses an architecture framework for automatic generation of architecture instances, and a software framework for mapping, scheduling, and generation of configuration for executing the application on the processor. The inputs to the tool automation framework are a set of architectures from the architecture generation framework and test-bench files from the software framework. The design of programming framework is not presented as it is beyond the scope of this thesis. The tool automation framework consists of an implementation phase and a simulation phase as explained in the following subsections.

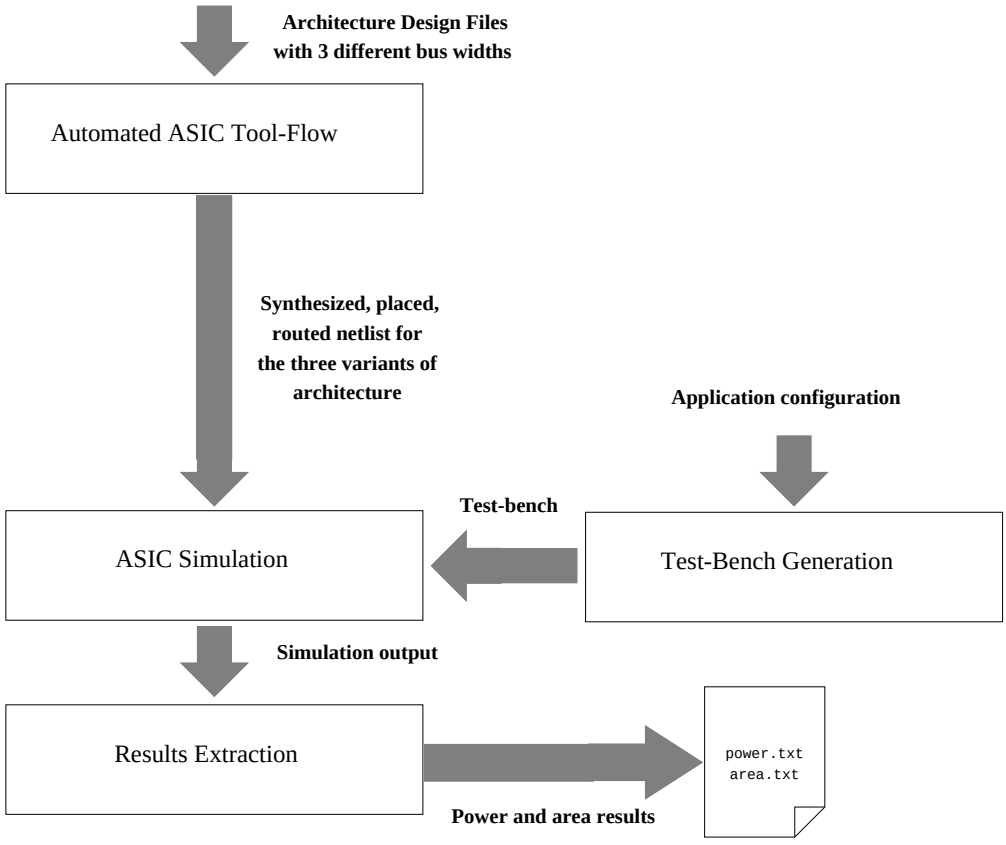

Figure 4.1: Tool Automation Framework

4.2.1 ASIC Implementation Framework

The tool automation framework is designed using Phyton scripts for automating the ASIC flow shown in Figure 4.2. The results of one stage of implementation are passed onto subse-

quent stages automatically without any manual intervention. The design files are given to the tool chain, and a gate-level netlist in Verilog that represents the layout of the processor architecture is obtained as output from the implementation framework. A standard cell based ASIC approach is utilized, and a 90-nm technology library from Synopsys that has capabilities to support leakage-aware design is used [44].

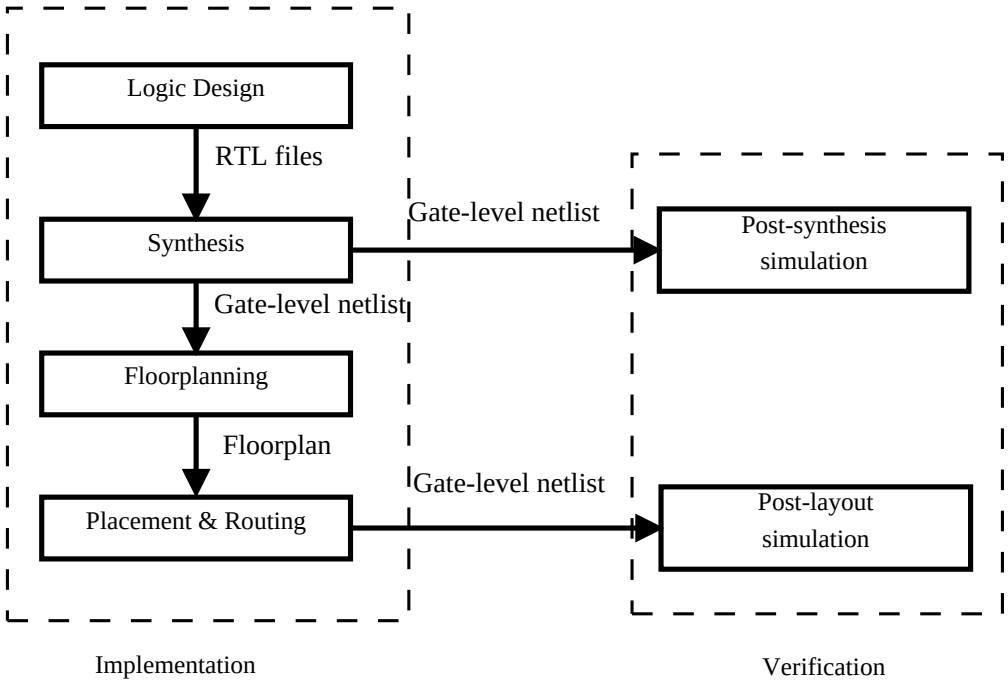

Figure 4.2: ASIC Process

The synthesis process transforms the RTL logic design into a structural design consisting of cells from the standard library. Synthesis optimizes the given RTL logic of the design using optimization constraints that affect the area, power, and timing of the chip. An objective of this thesis is to verify the functionality of the architecture, however, the optimization of the implementation is beyond the scope of this thesis. The Synopsys Design Compiler is used for synthesis of the architecture. The technology library used at synthesis stage is a database file consisting of logic cells of the library with timing and power models for cells. The design files, technology library, and the constraints file are inputs, and an equivalent gate-level netlist is generated at output. The commands for step-by-step execution of the

synthesis process are grouped in a Tool Command Language (TCL) file, and the framework invokes the synthesis tool with this TCL file to generate the results.

The gate-level netlist and the design constraints file output from the synthesis stage are inputs for the physical implementation stage. The Synopsys IC Compiler is used for floorplanning, automatic placement, and routing of the design. The logic library and the associated geometric library required for physical implementation are given to the IC compiler. A flow-chart representing the steps involved in physical synthesis is shown in Figure 4.3. A rectangular floorplan is created to accommodate all of the cells of the design, where the size of the floorplan is controlled by a utilization ratio. The exploration of physical implementation with respect to floorplan shape and size is beyond the scope of this thesis, and a fixed utilization factor is used for all architecture instances. The size of the floorplan, however, varies for different architecture instances, and is determined by the number and area of cells in the design.

The power rings and straps that form the power network of the chip are defined in the power-planning stage. The design is placed and pre-routed, connecting the power pins of all of the cells in the design to the power network. The pre-routed design is subject to clock-tree synthesis that synthesizes the clock network, and routes the clock pin connections for the registers in the chip. The resulting design is then routed. The time taken for clock-tree synthesis and the routing process do not differ significantly with variations in bus width of the architecture.

All of the above mentioned steps in physical implementation are done automatically by executing respective commands in the tool. The commands for the physical implementation process are grouped in a single TCL script file, and the framework invokes the Synopsys IC Compiler with this TCL. A gate-level netlist for the routed design, and the parasitic and timing information of the design are extracted from the output. The motivation for physical implementation is to analyze the area and power consumption associated with the functionality of the processor architecture. Therefore, the correction of design rule violations

and layout errors required for tape-out of the chip is not done, and is beyond the scope of this thesis.

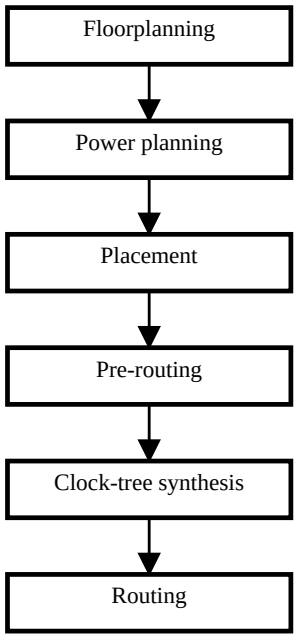

Figure 4.3: Physical Synthesis

4.2.2 ASIC Simulation Framework

The simulation framework executes an application on the processor by configuring the architecture prior to application execution, and providing input data during execution of an application. More precisely, the test-bench for simulation emulates the network to which the processor architecture would be integrated in an actual system. The configuration phase is done in bursts as discussed in Chapter 3, and the testbench gives input to the input FIFO in bursts. The burst length and the wait time after burst are calculated in the automatic testbench generation script. They are used as parameters in the Verilog code of testbench, emulating the behavior of the high-speed network. The simulation is performed at end of synthesis stage, and at end of physical implementation stage using the Synopsys VCS Compiler. The post-synthesis simulation involves simulating the gate-level netlist obtained from the Synopsys Design Compiler and Verilog model of the logic cells of the

technology library against the test bench. The post-layout simulation involves simulation of the gate-level netlist obtained from the Synopsys IC Compiler and Verilog model of logic cells of the technology library against the test bench. The gate-level netlist used for the post-layout simulation is back-annotated with parasitic information extracted from the Synopsys IC Compiler. The post-layout simulation results from the Synopsys VCS is used for power analysis of the design.

4.2.3 Results Extraction

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

4.3 UPF for Leakage-aware Implementation

This thesis deploys UPF-based techniques for implementing a leakage-aware design. The UPF is a standard for specifying the power intent in power-aware digital systems. The UPF provides a format for specification of power intent, where the power intent is an independent entity specified in a separate file. The UPF power intent commands specify creation of a power supply network, and a method for controlling this network for supplying power to logic cells in the chip. A subset of the UPF standard that supports a coarse-grained power

gating approach is utilized in this research project. A power domain is a logical collection of cells, instances, and blocks of the design that share the same power intent across the chip. All the elements of a power domain are connected to a common supply net. This common supply net could either be connected to an external power source or to the primary power source of the chip via a power gating switch. This power switch controls the power supply of the power domain by switching ON/OFF the common supply net. The elements of the chip that do not fall under any power domain are powered by the default supply net of the chip.

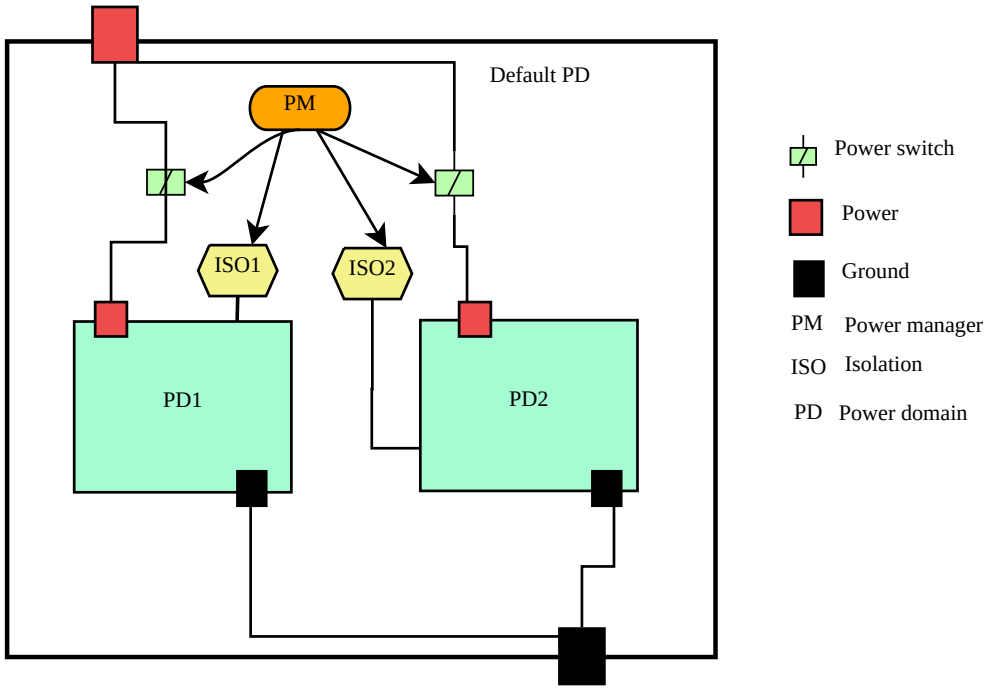

Figure 4.4: Sample Power-aware Design

The logic signals and nets of elements in a power domain are in an undefined state when the power supply is OFF. This undefined state might propagate to other parts of the chip. The UPF specification provides isolation techniques that protect the chip during the power-down state. Special logic cells called isolation cells are defined for logic signals that are output from power domains. The isolation cell passes on the desired value during the normal mode of operation, and clamps the logic signals to a predefined value during the power-down mode.

The design of the isolation strategy and the control signal for activating the isolation cells are specified in the UPF power intent file. A power manager in the RTL design file generates the control signals for the power switches and isolation cells. A sample top-level of a chip that has 2 power domains and switches for controlling them is shown in Figure 4.4.

The tool flow for implementing a power-aware design using UPF is shown in Figure 4.5. The Synopsys tool chain supports the UPF format in all stages of implementation of a leakage-aware design. A leakage-aware standard cell library consists of power-gating switches, and isolation cells required for implementing a power domain. A header switch is implemented by a PMOS transistor, and it controls the VDD of logic cells. A footer switch is implemented by a NMOS transistor, and it controls the GND of logic cells. The leakage power consumption of PMOS transistors is less than NMOS transistors. However, the size of a PMOS transistor is larger than an NMOS transistor to compensate for its low driving capability. The isolation circuitry deployed for a header switch based power domain is a simple pull-down transistor clamping the output signals to logic 0 state. The exploration of the advantages of header and footer power switches is beyond the scope of this thesis, and a header switch is chosen for controlling the power domains of the architecture.

The UPF power intent file is given as input along with RTL design files. The synthesis tool reads in the power intent file, and maps the isolation cells to logic signals for which the isolation strategy is specified. The power-switch is a physical cell controlling the power supply network. Therefore, it is not mapped until the power planning stage. The gate-level netlist from the synthesis stage consists of isolation cells. Verification of isolation strategy is done during post-synthesis simulation. The synthesis tool also generates an output UPF file corresponding to the input UPF, and this tool-generated UPF is passed onto subsequent stages.

In the physical implementation of a power-aware design, the floorplanning stage involves creation of voltage areas corresponding to power domains defined in the UPF specification. These voltage areas define a physical placement area for logic cells belonging to respective

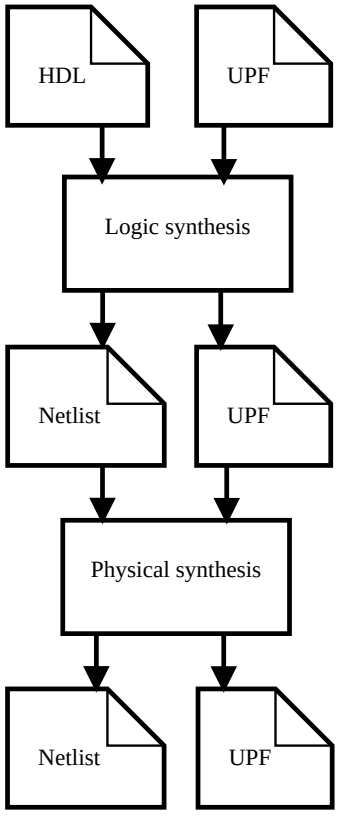

Figure 4.5: UPF Tool-flow

power domains. A sample floorplan with voltage areas is shown in Figure 4.6. Power switches are placed for controlling the power supply network of every voltage area. The power network synthesis of the tool is UPF-aware, and creates a supply network according to the power intent. The primary supply net of the chip is connected to the input of a power switch. The output of a power switch is connected to the common power supply net of a voltage area. The power manager in the wrapper of the functional unit generates control signals for switching ON/OFF the power switch. After power planning and routing, there are no changes in the subsequent steps of the physical implementation process. However, the clock-tree synthesis and routing stages are longer than usual due to additional restrictions imposed by voltage areas. Also, it was observed that the clock-tree synthesis, and the routing time for an architecture increases with the bus width. A gate-level Verilog netlist that has explicit power connections in the ports of logic cells is extracted from the IC Compiler at the end of

physical synthesis.

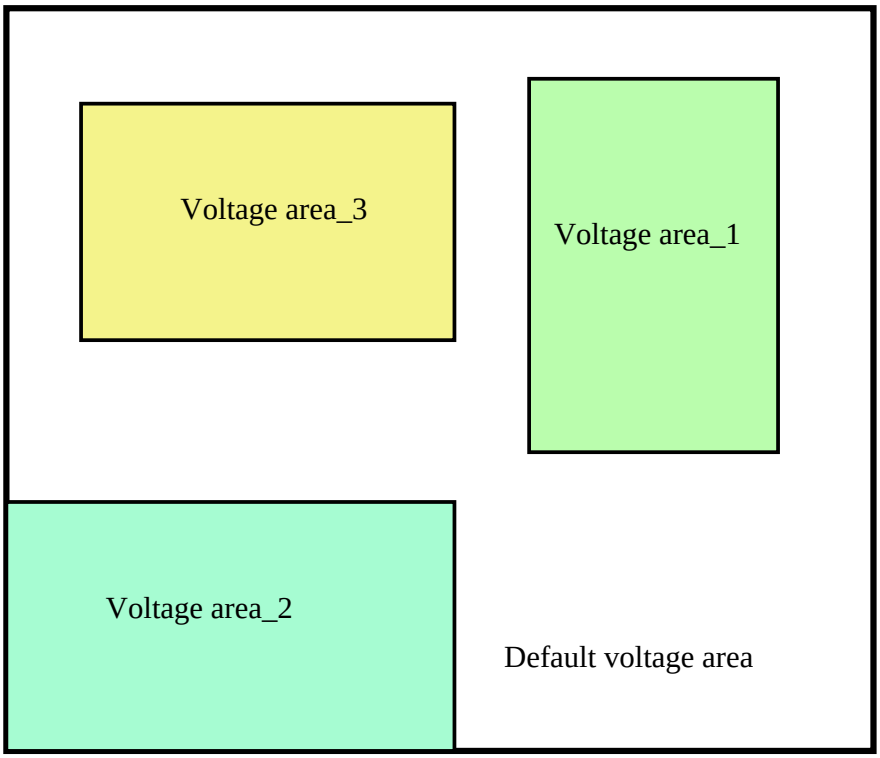

Figure 4.6: Sample Power-aware Floorplan

The verification of a power-aware design after placement and routing is two-fold. The functionality is verified together with the power intent. Post-layout simulation involves simulating the gate-level netlist and Verilog model of standard cells of the library against the testbench. The gate level netlist and Verilog model have explicit power and ground connections in the ports of their logic cells facilitating verification of the power intent. A power analysis is performed on the post-layout simulation output, and a detailed report of a hierarchical design is generated. A sample UPF design, and the tool flow for its implementation are presented in Appendix B.

Chapter 5

Results

This chapter presents the results of this thesis. Section 5.1 validates the design principles of the architecture by executing sample applications on various architecture instances. Section 5.2 discusses the power consumption and penalty associated with power-aware functional units. Section 5.3 presents the energy savings obtained from the power-aware design. Section 5.4 presents the power consumption of the functional units of the architecture that is visible to the programmer.

5.1 Validation of Design Principles

5.1.1 Experimental Design

The architecture is evaluated by the execution of sample applications. Three sample applications are chosen to evaluate the design principles of the architecture. Application 1 (or A1) is a temperature controller. Application 2 (or A2) is a 2-bit multiplier (Both the operands are 2 bits). Application 3 (or A3) is a hypothetical application. The applications are depicted as dataflow graph in Appendix C.

The processor architectures for executing the applications are depicted in Appendix D. Architecture 1 (or P1) and Architecture 2 (or P2) are targeted for A1. Architecture 3 (or P3) is targeted for the execution of A2 and A3. The limitations in the design of the processor architecture restricts the number of addressable functional units in the architecture. This is overcome by reusing the hardware units among application nodes, facilitated by a many-to-one mapping of application nodes to the functional units of the architecture. Such a scenario is validated by P2 and P3. All of the architectures are parameterizable, and various architecture instances are generated by varying the bus width.

The implementation framework performs synthesis, automatic placement, and routing of various architecture instances. The verification framework configures the applications onto the architectures, and generates the results required for the illustration of design principles of the architecture. The post-layout simulation output from the Synopsys VCS is a timing diagram depicting the execution of the functional units. This timing diagram is used for demonstrating the event-driven, concurrent, and pipelined execution of functional units. From this timing diagram, resource usage of input bus-interface, output bus-interface, and internal modules of the functional units are obtained.

5.1.2 Event-driven and Concurrent Execution

The post-layout simulation output for the execution of A1 on P1 with a bus width of 12 was analyzed, and the timing diagram of the functional units was obtained as shown in Figure 5.1. The execution of various modules of all the functional units of P1 are plotted against the time cycles. The following are the functional units of P1 : Timer (Tmr), Splitter (Spt), Analog to Digital Converter (Adc), special purpose Arithmetic and Logic Unit (Alu), Subtracter (Sub1 and Sub2), State machine (SME), and FIFO. IR represents the input bus-interface, OR represents the output bus-interface, IM represents the internal module of individual functional units. The bus activity (Bus act) is also shown in Figure 5.1.

The execution of a functional unit is depicted by the execution of its input bus-interface, in-

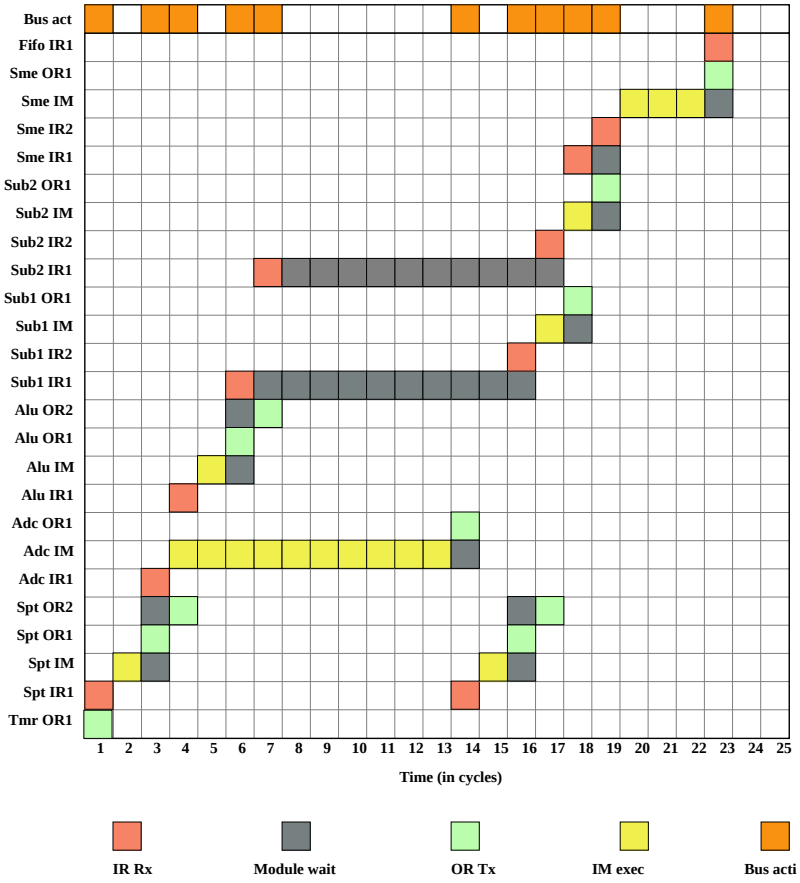

Figure 5.1: Execution of A1 on P1 with Bus Width of 12

ternal module, and output bus-interface. This figure shows that the execution of a functional unit is triggered by input events addressed to it. For an instance, when an A/D functional unit receives the timer event through splitter at execution cycle 3, it samples the analog channel, and starts the conversion from cycle 4. Similarly, the execution of the subtracter functional unit is triggered by inputs from the ALU and A/D converter units at cycle 6 and 16 respectively. The subtracter hardware starts execution at cycle 17 when all of its inputs are received. Moreover, the functional units of the architecture execute concurrently when their inputs are available. It is evident from Figure 5.1 that the A/D converter and ALU units execute concurrently in cycle 5 of the application execution. Thus, the execution of A1 on P1 illustrates the event driven and concurrent nature of the processor architecture.

5.1.3 Pipelined Execution of Functional Units

The execution on A1 on P2 verifies the reuse and pipelining of the functional units of the architecture. The Sub1 and Sub2 nodes of the application are mapped onto the same functional unit of the architecture. The post-layout simulation from the Synopsys VCS is analyzed. The execution of application node Sub2 is pipelined in the subtracter functional unit of the processor architecture. In the execution cycle 19, the output bus interface of the subtracter unit is used by Sub1 of the application node, and the internal module of the functional unit is used by Sub2 of the application node. Such a scenario illustrates the pipelined design of functional units.

The reuse and pipelining is more visible in applications in which the number of nodes are greater than the number of functional units in the architecture. Consider the execution of A2 on P3. Splitter, shifter, and state machine modules are reused extensively. The post-layout simulation output from the Synopsys VCS is analyzed. A 3-stage pipelining is visible for the execution of splitter nodes on the splitter functional unit. In cycles 31 and 32, the output bus-interface is used by the Split5 node, the internal module is used by the Split7 node, and the input bus-interface is used by the Split6 node. Thus, A2 serves as a suitable sample application for illustrating the pipelining feature of the functional units.

5.1.4 Flexibility of the Architecture

The functional units of the architecture are configurable by design, yielding a flexible processor architecture for executing multiple applications. Consider the execution of A2 and A3 on P3. In such a scenario, two different applications are executed on the same architecture. The functionality of the devices of the architecture is the same in both the applications. However, the configuration of the devices is different based on the application. The configurations of the functional units of P3 with bus width of 4 are summarized in Tables 5.1 and 5.2. This scenario illustrates the flexibility of the processor architecture.

Table 5.1: Configuration for A3

Modules	Destination	Delay	Num_used	Internal configuration
A/D converter	5,4	0,0	2	-
Constant	4	0	1	5
Shifter	0,7	2,0	2	-

Table 5.2: Configuration for A2

Modules	Destination	Delay	Num_used	Internal configuration
A/D converter	5,5	0,27	2	-
Constant	4,4	0,2	2	6,1
Shifter	5,5,3,3	0,2,5,0	4	-

5.1.5 Bus Width vs. Area and Performance

The architecture is parameterizable with respect to bus width. An increase in bus width reduces the end-to-end time required for executing an application. This is due to the reduction in the communication time required for transfer of events on the bus. Moreover, the packet formation logic at the bus-interfaces becomes simpler as the bus width increases. This is because the packet capture and the packet transfer blocks require reduced number of cells for their logic realization. Table 5.3 summarizes the impact of bus width on the area of the architecture, and execution time of the application. A2 is executed on P3. Various architecture instances are generated automatically from the architecture generation framework by varying the bus width. All of these architectures are automatically synthesized, placed, and routed by the implementation framework. The area results are obtained from the Synopsys IC Compiler tool integrated with the framework. The execution time represents the number of busy cycles in a single schedule period, and is obtained from the software framework.

Table 5.3: Bus Width vs. Area and Performance

Bus width	Area (μm^2)	Execution time (in cycles)
4	102999	99
6	102389	71
12	98480	47

The area and the per cycle energy cost of the following sections are obtained as follows. The area results of the individual modules are obtained from the Synopsys IC Compiler tool. The energy cost (per cycle) is calculated by multiplying average power (per cycle) with the clock period corresponding to 500KHz. The power consumption results are obtained from the Synopsys PrimeTime-PX with the aid of simulation results of physical synthesis.

5.1.6 Area and Energy Cost of a Template

As discussed in Chapter 3, the functional units are built by plugging the internal modules into the template. The bus-interfaces and the configuration registers form the template in a non-power-aware design. The components of the template, their area and energy cost (per cycle) are shown in Table 5.4. The wrapper configuration and the wrapper counter blocks form the configuration registers. It is evident that the bus-interfaces occupy more area, consuming more energy than the configuration registers. Moreover, the second input bus-interface is active in a functional unit only after the first input bus-interface receives the input. This contributes to the reduced energy cost of the second input bus-interface. The results presented in this section are for an architecture instance with a bus width of 4.

There are three different types of template. The Template-I consists of one input bus-interface and one output bus-interface, the Template-II consists of two input bus-interfaces and one output bus interface, and the Template-III consists of one input bus-interface and two output bus-interfaces. The area and energy cost of the templates are shown in Table 5.5. It is evident that the increase in number of bus-interfaces increases the area and the energy

Table 5.4: Area and Energy Cost of Template Components

Template Component	Area (μm^2)	Energy cost per cycle (pJ)
Input Bus-interface1	1781.5	10.5
Input Bus-interface2	1781.5	9.8
Output Bus-interface	1828.1	13.7
Wrapper Configuration	1114.6	6.7
Wrapper Counter	76.5	0.4

cost (per cycle) of the template. The energy cost was obtained for execution of A1 on P2.

Table 5.5: 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

5.1.7 Area and Energy Cost of a Template vs. Bus Width

The size and area of the bus-interfaces vary with the bus width of the architecture. An increase in bus width reduces the logic in the bus-interfaces as the packet formation logic becomes simple. However, there is an increase in logic in the bus-interfaces as they need to read/drive more wires with increase in bus width. The size of the driving cells is greater than the buffer cells used for reading the bus. The overall increase or decrease in the area, and the energy cost of the template are analyzed with the aid of the Tables 5.6 and 5.7. The acronyms IB and OB represent the input bus-interface and the output bus-interface respectively. The overall area and energy cost of the template decreases with the increase in the bus-width. The area of the input bus-interface decreases as the bus-width decreases because of the simplified logic for packet formation. The area of the output bus-interface

increases for bus width of 6 due to increase in number of logic cells (and their sizes) required to drive the bus. However, the area of output bus-interface for a bus-width of 12 decreases to a considerable extent. This is because the decrease in logic due to simplified packet formation is more than that of an increase in area due to increased number of logic cells required to drive the bus.

Table 5.6: Template-I Area vs. Bus Width

Bus Width	IB Area (μm^2)	OB 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

Table 5.7: Template-I Energy Cost vs. Bus Width

Bus Width	IB Energy Cost per cycle (pJ)	OB 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

5.1.8 Area and Energy Cost of a Template vs. Internal Units

This section presents the area and energy cost of functional units as a function of its template and internal modules. The energy cost was obtained for the functional units of the architecture for the execution of A1 and A2 on P2 and P3 respectively. It is evident from Table 5.8 that the template forms a major portion of the total area and energy cost of the functional units. Every functional unit has a different type of template depending on the number of bus-interfaces required for its functionality. Therefore, the type of the template is also specified in the table. The State Machine Engine functional unit is represented as SME.

The internal module of a state machine is bulkier than its template. The results shown in Table 5.8 are obtained for an architecture instance with bus width of 4.

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)
A/D converter	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

5.1.9 FIFO Size vs. Template Size

As template size could be used as a unit of measure for area occupied by functional units of the architecture, it is also used to study the area occupied by the FIFO. The FIFO was designed in a parameterizable fashion controlling its depth. The area of FIFO for various depths is shown in Table 5.9. The width of the FIFO is the same, and is 12-bits (same as packet length of the architecture). The size of the FIFO increases as its depth increases. The area of Template-I is considered for comparison. This experiment is done for an architecture instance with a bus width of 12. The area results were extracted after physical synthesis from the Synopsys IC Compiler tool. The results show that the FIFO's internal module is bulkier than the template of the architecture.

Table 5.9: FIFO Size vs. Template Size

Depth of FIFO	Area of FIFO (μm^2)	Area of the Template (μm^2)
8	5382.2	4355.3
16	10084	4355.3
32	18868	4355.3
64	36547	4355.3

5.2 Power Consumption of Functional Units

This section presents the power consumption results of functional units for the execution of A1 on P2. The post-layout simulation output of the architecture is extracted from the verification framework. The timing and parasitic information of the architecture are extracted from the Synopsys IC Compiler by the implementation framework. These inputs from the framework are given to the Synopsys PrimeTime-PX tool. Power consumed by the functional units, and the processor architecture for every cycle of an application execution are obtained by invoking the automation scripts. The cycle-by-cycle power results from the tool are grouped for idle and non-idle states of the functional units, and an average power consumption during these states is obtained. A functional unit is in idle state when there is no activity in the event bus, and when all of its modules are idle. In non-idle state, there is activity in the bus, triggering the execution of input bus-interfaces. Also, a functional unit is in non-idle state when any of its modules is in execution phase.

The leakage power consumption of the functional units such as A/D converter, splitter, and subtracter is shown in Table 5.10. The internal power and switching power are the components of dynamic power, and are shown in Tables 5.11 and 5.12. The dynamic power is dependent on the switching activity of the modules in a functional unit, and therefore, it is presented for both the idle and non-idle states. The leakage power of a functional unit is independent of the switching activity, and the values are more or less same for both the idle and non-idle states. It is evident that the dynamic power consumption is less in idle state,

as there is no switching activity. From the results, it is known that the leakage power is dominant over the dynamic power consumption. For instance, the total power consumption of an A/D functional unit in the idle and non-idle states are 24 μW and 24.5 μW respectively. In the non-idle state, $\approx$ 80% of the total power consumption is the leakage power. Also, in the idle state, $\approx$ 81% of the total power consumption is the leakage power. It is because of the sub-micron standard cell ASIC library used for the implementation of the architecture. The power results are determined by the power model of the standard cell library, and a change in the ASIC library would yield different results. Also, the accuracy of power results depends on the correctness of the power model in the standard cell library.

Table 5.10: Leakage Power of Functional Units

A/D	Splitter	Subtracter
19.6 μW	25.4 μW	26.4 μW

Table 5.11: Dynamic Power of Functional Units: Idle State

	A/D	Splitter	Subtracter
Internal power	4.43 μW	6.14 μW	5.92 μW
Switching power	0	0	0

Table 5.12: Dynamic Power of Functional Units: Non-idle State

	A/D	Splitter	Subtracter
Internal power	4.83 μW	7.30 μW	7.00 μW
Switching power	78.7 nW	182 nW	222 nW

To reduce leakage power consumption, power-aware design techniques were deployed for the functional units of the architecture. As discussed in Chapter 3, the internal and the output bus-interface modules are powered-down. The number of power-states of a functional unit is 3. They are (OFF,OFF) when these modules are powered-down, (ON,ON) when these

modules are powered-on, and (OFF,ON) when the internal module is powered-down and the output bus-interface is powered-up. The (ON,OFF) state is not supported in the design. A UPF-based power-aware design is implemented, and the power results are obtained.

Table 5.13: Leakage Power of Functional Units in Power-aware Design

States	A/D	Splitter	Subtracter
(OFF,OFF)	14.8 μW	16.7 μW	21.6 μW
(ON,ON)	22.8 μW	29.0 μW	29.7 μW
(OFF,ON)	19.7 μW	26.3 μW	27.5 μW

Table 5.14: Leakage Power Comparison

Modules	(OFF,OFF)	(ON,ON)	(OFF,ON)
A/D	-24.48%	$+16.32\%$	$+0.5\%$
Splitter	-34.25%	$+14.17\%$	$+3.5\%$
Subtracter	-18.16%	$+12.25\%$	$+2.6\%$

The leakage power consumption of a power-aware design for various power-states is shown in Table 5.13. A comparison of leakage power for a power-aware and a non-power-aware architecture is shown in Table 5.14. The leakage power consumption of the functional units are reduced when their power state is (OFF,OFF). There is an increase in the leakage power when both the modules are powered-on. This is because of the additional logic for the power management included in the wrapper of a functional unit. The number of cycles that both of these modules are ON is less, and therefore, this power penalty is not a critical factor. The analysis of (OFF,ON) state has to be considered on a functional unit basis. In the A/D functional unit, the number of cells in the internal module is much greater than the number of cells in the power manager. Therefore, switching off the internal module has more impact, and the increase in power consumption is only 0.5%. In the subtractor unit, the number of cells in its internal and power manager modules are more or less the same. Therefore, switching off the internal module of the subtractor when its output bus-interface

is ON increases the leakage power consumption by 2.6%. The increase is more pronounced in the splitter unit as the number of output bus-interfaces is 2 when compared to other units. Another feasible power state for modules with 2 output bus-interfaces is (OFF,PARTIALLY ON). In this state, the internal module is OFF, one of the output bus-interface is OFF, and the second output bus-interface is ON. The leakage power consumed by the splitter in such a state is 21.6 μW, and the power savings is 14.9%.

Table 5.15: Dynamic Power of Functional Units in Power-aware Design: Idle State

	A/D	Splitter	Subtracter
Internal power	1.63 μW	1.63 μW	3.00 μW
Switching power	0 W	0 W	0 W

Table 5.16: Dynamic Power of Functional Units in Power-aware Design: Non-idle State

	A/D	Splitter	Subtracter
Internal power	4.29 μW	6.13 μW	6.64 μW
Switching power	107 nW	287 nW	466 nW

The dynamic power consumption of a power aware design for the idle and non-idle states are shown in Tables 5.15 and 5.16. The leakage-aware techniques reduce the internal switching power of a functional unit when in idle state. The power savings for the A/D converter and subtracter are 22.5% and 49.3%. The power savings is more pronounced for the splitter unit, and is 73%. This is because the number of output bus-interfaces is 2. There is an increase in the dynamic power consumption (in the non-idle states) of the functional units. This is due to the additional power manager module added in the wrapper of a functional unit.

5.2.1 Penalty Associated with the Power-aware Design

There is an increase in area and execution time of the functional units due to the power management circuits. The area and per cycle energy cost of the template for a power-

aware and a non-power-aware approach is shown in Table 5.17. The values are shown for an architecture instance with bus width of 12. The template includes the bus-interfaces and the wrapper registers in a non-power-aware design. The power manager modules are added to the template in a power-aware design. The hierarchical area results are obtained from the Synopsys IC Compiler. The cycle-by-cycle power results are obtained from the power analysis tool, and the average energy cost (per cycle) is calculated for the frequency of 500 KHz. The output registers in the wrapper are powered-down when not in use, and therefore, the energy cost of a power-aware template varies. The value for both the scenarios is shown in Table 5.17. The simplest template with one input and one output bus-interfaces, and one set of configuration registers is taken into consideration. The acronym OB represents the output bus-interface. The per cycle energy cost is increased by 3.42 units when the output bus-interface is ON. Also, the per cycle energy cost of a template is decreased by 5 units when the output bus-interface is OFF.

Table 5.17: Comparison of Area and Energy Cost of Template

	Area	Energy cost (per cycle)
Power-aware (OB ON)	4933.1 μm^2	32.62 pJ
Power-aware (OB OFF)	4933.1 μm^2	24.2 pJ
Non-power-aware	4355.3 μm^2	29.2 pJ

The powered-down modules of a functional unit take 2 cycles for power-up. This increases the execution time of a function unit by 2. This increase is not significant in complex applications that reuse the functional units in the architecture. The power-down of modules of a functional unit may not be required if the next set of inputs are readily available. The total execution time of A1 on P1 is increased by 15 cycles for a power-aware design. The power consumed by the A/D converter in the power-up cycle is 28.3 μW. The total power consumed by the A/D converter in the non-idle state 27.5 μW. Therefore, the power consumed in the power-up cycles is greater than the average power consumed in the non-idle states of a functional unit.

5.3 Total Energy Consumption

This section presents the power and energy consumption of P2 for the execution of A1. All of the functional units of the architecture are power managed, and they are in different power states during every cycle of execution. The power states and the power consumed by a few functional units of this architecture were discussed in the previous section. The application required the sampling of analog channel at the rate of ≈ 4times/ms. The configuration of the timer and the input FIFO were derived from this specification. The execution clock is 500KHz. The cycle-by-cycle power analysis results are grouped for busy and free cycles. The average power consumption in busy and free cycles for a non-power-aware and power-aware design is shown in Tables 5.18 and 5.19 respectively. The number of busy and free cycles is shown in parenthesis. The power-up latency of the functional units increases the number of busy cycles by 15 units in the power-aware design.

Table 5.18: Total Power and Energy: Non-power-aware Design

	Busy Cycle (24)	Free Cycle (106)
Switching Power	4.15 μW	3.66 μW
Internal Power	54.3 μW	51.9 μW
Leakage Power	208 μW	208 μW
Total Power	266 μW	263 μW
Total Energy	12.76 nJ	55.75 nJ

Table 5.19: Total Power and Energy: Power-aware Design

	Busy Cycle (39)	Free Cycle (91)	Free Cycle (Input OFF)
Switching Power	4.52 μW	3.78 μW	3.70 μW
Internal Power	27.5 μW	20.9 μW	16.1 μW
Leakage Power	161 μW	149 μW	126 μW
Total Power	193 μW	173 μW	146 μW
Total Energy	15.05 nJ	31.48 nJ	26.67 nJ

The overall energy consumption of a non-power-aware and a power-aware architecture for

one schedule period of application execution are calculated. The energy consumption is given by the following formula.

$$E_{sched} = (P1_{avg}) * \tau * N1 + (P2_{avg}) * \tau * N2 \; where,$$

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

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

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

τ *is the processor clock period in seconds*,

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

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

The total energy consumed by a power-aware design is $\approx$ 32% less than that of a non-power-design. The reduction in energy consumption is $\approx$ 39% if the input bus-interfaces are powered down in the free cycles. The limitations in the physical synthesis tool restricted the number of power domains, therefore, the input bus-interfaces were not powered-down in the free cycles. The internal power reduces to 0, and the leakage power reduces by a factor of $\approx$ 7 times when a module is powered-down. This was observed with the output bus-interface module. The same factors were used in estimating the power consumption if the input bus-interfaces were powered-down in the free cycles. The reduced power consumption of all the input bus-interfaces are estimated. The power-penalty associated with the power manager circuitry for the input bus-interface is assumed to be same as that of an output bus-interface. Every ouput bus-interface in the architecture has a power penalty of $\approx$ 1.1 μW. However, one power manager circuitry manages all the input bus-interfaces, and the total power penalty is $\approx$ 1.1 μW. This was added to the power consumption in the free cycles. Thus, the estimated power values for the input bus-interface OFF scenario was calculated considering both the power savings and the power penalty. The estimated power comsumption is calulated as

follows:

$P_{est} = (P_{obt}) - ((P_{inp}) * (number\ of\ input\ bus - interfaces)) + (P_{plty})\ where,$

P_{est} *is the estimated power in a free cycle when inputs are OFF,*

P_{obt} *is the average power consumption in a free cycle,*

P_{inp} *is the decrease in power consumption of a power − aware input bus − interface,*

P_{plty} *is the power penalty of a power − aware input bus − interface,*

The energy consumption in the free cycles could further be reduced if it was possible to switch OFF the configuration registers. The number of configuration registers depends on the number of times that a module is reused. The minimum average power consumed by the configuration registers in a functional unit is 3.8 μW. The power consumption of configuration registers increases with the reuse of a functional unit [43]. Moreover, the internal modules of a timer unit and the state machine unit are not powered-down, and they contribute to the power consumption in the free cycles.

5.4 Power Consumption Visible to the Programmer

The number of feasible power states of a functional module is 3 during execution of an application. As the processor architecture is deterministic, the power-states of a functional unit is visible to the programmer prior to execution of an application on the architecture. If a power-state based power-model is generated for all the functional units of the architecture, the power consumption for executing an application is visible to the programmer. This power model could be represented as a look-up table, and the power consumption could be estimated by the programmer after static scheduling in the software [43].

A partial attempt has been made to generate the power-model for selected functional units of the architecture. The leakage power model for the functional units such as A/D converter, and splitter is listed in Table 5.20. The leakage power is independent of the switching activity and frequency of operation of the architecture. The model is constructed by executing applications on various architecture instances. The leakage power data is collected for all the power states. An average of the values is taken for the construction of a power model. The number of sample architectures considered is 4. The number of functional units in the architecture were increased from 3 to 8. There was no significant increase in the leakage power consumption of functional units in all these different architectures. This is due to the fact that the logic realizations of bus-interfaces did not differ to a significant extent when the number of functional units connected to an event bus was increased. The experiment was performed for architecture instances with a bus-width of 12-bits.

Table 5.20: Leakage Power Model of Functional Units

Internal Module	Output Interface	A/D	Splitter
OFF	OFF	13 μW	18.1 μW
ON	ON	21 μW	27.8 μW
OFF	ON	17.8 μW	22.8 μW

The dynamic power model depends on the switching activity and the frequency of operation. This could be constructed by executing sample applications at multiple frequencies. The frequency of operation of the implemented processor architecture is limited by the standard cell library, and a wide range of operation is not possible. Therefore, dynamic power modeling of the functional units was not feasible.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, a simple version of the processor architecture was designed and implemented as an ASIC. The processor and the software framework together facilitated the execution of simple applications, aiding in the illustration of the design principles. The parameterizable design approach of the architecture, and its functional units were helpful in the reuse of the same logic blocks across various architecture instances. The configurability of the functional units of the architecture facilitated execution of different applications on the same architecture hardware.

In this thesis, certain limitations with the design approach were observed. In the earlier design approach, each of the functional units supported the execution of exactly one application node. This limited the application space of the architecture as the design supported only eight addressable units. This was solved by reusing a functional unit among the similar nodes of an application. This was done by modifying the design of the configuration registers. A set of registers were used in place of a single register in the configuration modules. This approach, however, increased the area and energy cost of template of the functional

units [43].

Another issue that was observed with the earlier design was related to excess leakage power consumption. This was partly because of the deep sub-micron technology library used for the ASIC implementation, and partly because of the functional units being in an idle state for more cycles in a single schedule period. A leakage-aware design approach was deployed to reduce the static power consumption of the functional units. This approach, however, incurred an increase in the area of the chip. Also, there was a power penalty during non-idle states because of the power manager circuits. The total energy consumption for executing an application on a power-aware architecture was less than that of a non-power-aware architecture.

6.2 Future Work

The functional units of the architecture are simple, and they were designed to execute simple test applications. The exploration of more complex functional units such as filter is needed to expand the application space of the architecture. It is not possible to execute applications that have more than eight different functional nodes. The architecture needs to be expanded to support complex applications by modifying its packet structure. A modified packet structure with increased number of bits is required for extending the address space of the functional units. Thus, the packet structure and its length is another area that needs exploration. When the number of functional units are increased, connecting all of them to a single event bus, and sharing the bus might not yield required performance. Therefore, exploration of architectures with multiple event buses is needed. The FIFOs are designed in a parameterizable fashion, controlling its depth. The right depth of FIFO depends on the speed of the network, the speed of the processor, and the distribution of incoming packets from the network. Exploration of these factors to determine the depth of the FIFO is required in future.

The power-aware architecture supports the power-down and power-up of modules of a functional unit during execution. The wrapper registers are not powered-down. However, the static power consumption associated with the always-on modules could be reduced by deploying retention flops. The UPF power intent has provisions for such an approach, and this methodology could be explored in future versions of the architecture. The retention flops are constructed by high-threshold transistors, that reduce the static power consumption. Furthermore, the use of low power flash memories to store configurations would have an impact on the overall power consumption, and thus, it is a scope for future work.

Bibliography

[1] C. Linti, H. Horter, P. Osterreicher, and H. Planck, "Sensory Baby Vest for the Monitoring of Infants," *International Workshop on Wearable and Implantable Body Sensor Networks*, vol. 0, pp. 135–137, 2006.

[2] M. Chandra, M. T. Jones, and T. L. Martin, "E-Textiles for Autonomous Location Awareness," *Wearable Computers, IEEE International Symposium*, vol. 0, pp. 48–55, 2004.

[3] J. Edmison, M. Jones, T. Lockhart, and T. Martin, "An E-textile System for Motion Analysis," in *International Workshop on New Generation of Wearable Systems for eHealth*, 2003, pp. 215–223.

[4] 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.

[5] C. M. Zeh, "Softwear: A Flexible Design Framework for Electronic Textile Systems," Master's thesis, Virginia Polytechnic and State University, April 2006.

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

[7] L. Yang, M. Ji, Z. Gao, W. Zhang, and T. Guo, "Design of Home Automation System Based on ZigBee Wireless Sensor Network," *International Conference on Information Science and Engineering*, vol. 0, pp. 2610–2613, 2009.

[8] N. Wang, N. Zhang, and M. Wang, "Wireless Sensors in Agriculture and Food Industry–Recent Development and Future Perspective," *Computers and Electronics in Agriculture*, vol. 50, no. 1, pp. 1 – 14, 2006.

[9] L. Chen, Z. Chen, and S. Tu, "A Realtime Dynamic Traffic Control System Based on Wireless Sensor Network," in *ICPPW '05: Proceedings of the 2005 International Conference on Parallel Processing Workshops*. Washington DC, USA: IEEE Computer Society, 2005, pp. 258–264.

[10] R. Shenoy, "Design of E-Textiles for Acoustic Applications," Master's thesis, Virginia Polytechnic and State University, September 2003.

[11] D. Graumann, G. Raffa, M. Quirk, B. Sawyer, J. Chong, M. Jones, and T. Martin, "Large Surface Area Electronic Textiles for Ubiquitous Computing: A System Approach," in *MOBIQUITOUS '07: Proceedings of the 2007 Fourth Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous)*. Washington DC, USA: IEEE Computer Society, 2007, pp. 1–8.

[12] J. Chong, "Activity Recognition Processing in a Self-Contained Wearable System," Master's thesis, Virginia Polytechnic and State University, September 2008.

[13] Arvind and D. E. Culler, "Dataflow Architectures," in *Annual review of computer science*, vol. 1. Palo Alto, CA, USA: Annual Reviews Inc., 1986, pp. 225–253.

[14] J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data-flow Processor," *SIGARCH Computer Architecture News*, vol. 3, no. 4, pp. 126–132, 1974.

[15] K. Arvind and R. S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Transaction on Computers*, vol. 39, no. 3, pp. 300–318, 1990.

[16] I. P. Radivojevic and J. Herath, "Executing DSP Applications in a Fine-Grained Dataflow Environment," *IEEE Transactions on Software Engineering*, vol. 17, no. 10, pp. 1028–1041, 1991.

[17] G. Liao, G. R. Gao, and V. K. Agarwal, "A Dynamically Scheduled Parallel DSP Architecture for Stream Flow Programming," *Journal of Microcomputer Applications*, vol. 17, no. 2, pp. 171–196, 1994.

[18] J. Sosa, J. Boluda, F. Pardo, and R. Gmez-Fabela, "Change-driven Data Flow Image Processing Architecture for Optical Flow Computation," *Journal of Real-Time Image Processing*, vol. 2, no. 4, pp. 259–270, 2007.

[19] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers, "The WaveScalar Architecture," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 2, pp. 1–54, 2007.

[20] M. B. Rutzig, A. C. S. Beck, and L. Carro, "Transparent Dataflow Execution for Embedded Applications," in *ISVLSI '07: Proceedings of the IEEE Computer Society Annual Symposium on VLSI*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 47–54.

[21] J. Carlstrom and T. Boden, "Synchronous Dataflow Architecture for Network Processors," *IEEE Micro*, vol. 24, no. 5, pp. 10–18, 2004.

[22] D. Gay, P. Levis, and D. Culler, "Software Design Patterns for TinyOS," in *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. New York, NY, USA: ACM, 2005, pp. 40–49.

[23] M. Hempstead, G.-Y. Wei, and D. Brooks, "An Accelerator-based Wireless Sensor Network Processor in 130nm CMOS," in *CASES '09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*. New York, NY, USA: ACM, 2009, pp. 215–222.

[24] A. Mota, L. B. Oliveira, F. F. Rocha, R. Riserio, A. A. F. Loureiro, C. J. N. Coelho Jr., H. C. Wong, and E. Nakamura, "WISENEP: A Network Processor for Wireless Sensor Networks," in *ISCC '06: Proceedings of the 11th IEEE Symposium on Computers and Communications*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 8–14.

[25] R. Gerndt and R. Ernst, "An Event-Driven Multi-Threading Architecture for Embedded Systems," in *CODES '97: Proceedings of the 5th International Workshop on Hardware/-Software Co-Design*. Washington DC, USA: IEEE Computer Society, 1997, p. 29.

[26] V. N. Ekanayake, C. Kelly, IV, and R. Manohar, "BitSNAP: Dynamic Significance Compression for a Low-Energy Sensor Network Asynchronous Processor," in *ASYNC '05: Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 144–154.

[27] R. Bittner and P. Athanas, "Wormhole Run-time Reconfiguration," in *FPGA '97: Proceedings of the 1997 ACM fifth international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM, 1997, pp. 79–85.

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

[29] L. Nazhandali, B. Zhai, J. Olson, A. Reeves, M. Minuth, R. Helfand, S. Pant, T. Austin, and D. Blaauw, "Energy Optimization of Subthreshold-Voltage Sensor Network Processors," *SIGARCH Computer Architecture News*, vol. 33, no. 2, pp. 197–207, 2005.

[30] B. Zhai, S. Pant, L. Nazhandali, S. Hanson, J. Olson, A. Reeves, M. Minuth, R. Helfand, T. Austin, D. Sylvester, and D. Blaauw, "Energy-efficient subthreshold processor design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 8, pp. 1127–1137, 2009.

[31] M. Hempstead, N. Tripathi, P. Mauro, G.-Y. Wei, and D. Brooks, "An Ultra Low Power System Architecture for Sensor Network Applications," *SIGARCH Computer Architecture News*, vol. 33, no. 2, pp. 208–219, 2005.

[32] M. Seok, S. Hanson, Y.-S. Lin, Z. Foo, D. Kim, Y. Lee, N. Liu, D. Sylvester, and D. Blaauw, "The Phoenix Processor: A 30pW Platform for Sensor Applications," in *IEEE Symposium on VLSI Circuits*, 18-20 2008, pp. 188 –189.

[33] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage Current: Moore's Law Meets Static Power," *Computer*, vol. 36, no. 12, pp. 68–75, 2003.

[34] M. Hempstead, G.-Y. Wei, and D. Brooks, "Architecture and Circuit Techniques for Low-throughput, Energy-constrained Systems Across Technology Generations," in *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems.* New York, NY, USA: ACM, 2006, pp. 368–378.

[35] C. LaFrieda and R. Manohar, "Reducing Power Consumption with Relaxed Quasi Delay-Insensitive Circuits," in *ASYNC '09: Proceedings of the 2009 15th IEEE Symposium on Asynchronous Circuits and Systems (async 2009).* Washington, DC, USA: IEEE Computer Society, 2009, pp. 217–226.

[36] J. Yantchev, C. Huang, M. Josephs, and I. Nedelchev, "Low-latency Asynchronous FIFO Buffers," in *Proceedings of Second Working Conference on Asynchronous Design Methodologies*, 30-31 1995, pp. 24 –31.

[37] W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava, "Component-based Design Approach for Multicore SoCs," in *DAC '02: Proceedings of the 39th annual Design Automation Conference.* New York, NY, USA: ACM, 2002, pp. 789–794.

[38] S. Bailey, G. Chidolue, and A. Crone, "Low Power Design and Verification Techniques," http://www.low-powerdesign.com/Low_Power_WP_9-13-07.pdf, 2007.

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

[40] "Unified Power Format (UPF) Standard," http://www.unifiedpowerformat.com/images/UPF.v1.0_Standard.pdf, February 2007.

[41] "IEEE Standard for System Verilog - Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800-2005*, pp. 1–648, 2005.

[42] J. Bromley and M. Smith, "A Users Experience with SystemVerilog," http://www.doulos.com/knowhow/sysverilog/snug04europe/snug04_bromley_smith_paper.pdf, 2004.

[43] K. Lakshmanan, "Design of an Automation Framework for a Novel Data-Flow Processor Architecture," Master's thesis, Virginia Polytechnic and State University, 2010, (to be published).

[44] G. Delp, J. Biggs, and S. Jadcherla, "Design and Verification of Low Power SoCs," http://www.accellera.org/home/20090316_ISQED_LowPowerPrintAll.pdf, 2009.

Appendix A

SystemVerilog Models

This section presents the SystemVerilog model of functional units such as Shifter and an A/D converter. From their RTL models, it is evident that these modules have a common wrapper structure, however, their internal modules differ. An abstract top-level represented as a hierarchy of the wrapper and internal modules is shown below.

SystemVerilog model 1

```
//Top level for the shifter unit
module shifter #
(parameter bit [2:0] MOD_ID  = 3'b000,
parameter int BUS_WIDTH  = 4,
parameter int PACKET_WIDTH  = 12,
parameter int MAX_COUNT  = 3,
parameter int NO_OF_OPERANDS = 2,
parameter int NUM_OUTPUTS = 1,
parameter int WR_CNT_WIDTH = 1)
(...);

//Wrapper modules
pwr_mgmt_int shift_pwr_mgr_int (...);
inp_reg_mod # ( MOD_ID ,BUS_WIDTH ,PACKET_WIDTH ,MAX_COUNT) inp_reg1 (...);
inp_reg_mod # ( MOD_ID ,BUS_WIDTH ,PACKET_WIDTH ,MAX_COUNT) inp_reg2 (...);
wr_config_block # (WR_CNT_WIDTH, NUM_OUTPUTS) wrapper_config (...);
wrapper_cnt_mod # (WR_CNT_WIDTH) wr_cnt_mod1 (...);
out_reg_mod # (PACKET_WIDTH,BUS_WIDTH,MAX_COUNT) output_reg (...);

//Internal module
shifter_int_mod # ( NO_OF_OPERANDS ) int_mod (...);


endmodule
```

SystemVerilog model 2

```
//Top level for the A2D converter unit
module adc_converter #
(parameter bit [2:0]  MOD_ID  = 3'b000,
paraameter int BUS_WIDTH  = 4,
parameter int PACKET_WIDTH  = 12,
parameter int MAX_COUNT  = 3,
parameter int NO_OF_OPERANDS = 1,
parameter int NUM_OUTPUTS = 1,
parameter int WR_CNT_WIDTH = 1,
parameter int RESOLUTION_BITS = 8)
(...);

//Wrapper modules
pwr_mgmt_int adc_pwr_mgr_int (...);
inp_reg_mod # ( MOD_ID ,BUS_WIDTH ,PACKET_WIDTH ,MAX_COUNT) inp_reg1 (...);
wr_config_block # (WR_CNT_WIDTH, NUM_OUTPUTS) wrapper_config (...);
wrapper_cnt_mod # (WR_CNT_WIDTH) wr_cnt_mod1 (...);
out_reg_mod # (PACKET_WIDTH,BUS_WIDTH,MAX_COUNT) output_reg1 (...);

//Internal modules
adc_internal_module # (NO_OF_OPERANDS,RESOLUTION_BITS) adc1 (...);
analog_model # (RESOLUTION_BITS) Analog_blk1(...);

endmodule
```

Appendix B

Tutorial for UPF

This section provides a sample UPF power intent file, and describes the process for a power-aware design. This tutorial is provided as a guidance for those who would be expanding this research work in future. An architecture instance with power management for the A/D converter is considered. The following is the step-by-step approach for designing and implementing a power-aware functional unit.

1. A SystemVerilog model of the architecture is created as a hierarchy of its functional units. The A/D converter functional unit has a power manager module in its wrapper

2. The UPF power intent files are created in a hierarchical fashion. The top-level of the main UPF file has a hierarchy of power domains. The UPF files for A/D converter and the architecture are given in the UPF models in the following sections

3. The synthesis is performed using the Synopsys Design Compiler tool. The RTL code and the UPF power intent are given as inputs. The tool environment is set up with the required synthesis library paths. A step-by-step process for synthesis is given in the tool commands script for synthesis

4. The gate-level Verilog netlist from the synthesis tool is simulated against the testbench

using the Synopsys VCS tool. The isolation circuits and power-on reset functionality are verified at this stage. The simulation commands are also given the tool commands script for simulation

5. The gate-level Verilog netlist, the tool-generated UPF, and the design constraints file from the Synopsys Design Compiler are given to the Synopsys IC Compiler tool. This tool has to be invoked in 64-bit mode. A step-by-step process for the physical synthesis is given in the tool commands script for physical synthesis

6. A gate-level Verilog netlist that has explicit power and ground ports in all of its cells is obtained from the Synopsys IC Compiler. This file is simulated against the testbench, and the power intent is verified in this simulation. The simulation commands are given in the scripts in the following sections

7. The tool-generated UPF file from the Synopsys IC Compiler, the post-layout simulation output from the Synopsys VCS, and a gate level Verilog netlist (that has no explicit power and ground connections) from the Synopsys IC Compiler are given to the power analysis tool. The script for power analysis is also provided in the subsequent sections

Upf model 1 Upf for A/D converter internal module

```
####Upf script for a2d internal module

###creating a power domain
create_power_domain adc_ctrl -include_scope
###creating supply nets of the domain
create_supply_net VDDG  -domain adc_ctrl
create_supply_net VSS  -domain adc_ctrl
create_supply_net adc_VDD -domain adc_ctrl
###creating supply ports of the domain
create_supply_port VDDG  -domain adc_ctrl
create_supply_port VSS  -domain adc_ctrl
###connecting nets to ports
connect_supply_net VDDG -ports VDDG
connect_supply_net VSS -ports VSS
###Primary power nets for the power domain
set_domain_supply_net adc_ctrl
-primary_power_net adc_VDD  -primary_ground_net VSS
###Power switch definition
create_power_switch SW -domain adc_ctrl -output_supply_port {vout adc_VDD}
-input_supply_port {vin VDDG}
-control_port {adc_ctrl_port adc_pwr_ctrl}
-on_state {ON vin {adc_ctrl_port}}
-off_state {OFF {!adc_ctrl_port}}
```

Upf model 2 Upf for A/D converter output bus-interface module

```
####Upf script for a2d output_register1

###Create a power domain for output register
create_power_domain adc_op1 -include_scope
###create supply nets
create_supply_net VDDG  -domain adc_op1
create_supply_net VSS  -domain adc_op1
create_supply_net adc_VDD1 -domain adc_op1
###create supply nets
create_supply_port VDDG  -domain adc_op1
create_supply_port VSS  -domain adc_op1
###connect supply nets to ports
connect_supply_net VDDG -ports VDDG
connect_supply_net VSS -ports VSS
###Primary supply nets for the power domain
set_domain_supply_net adc_op1
-primary_power_net adc_VDD1
-primary_ground_net VSS
###Power switch definition
create_power_switch SW1 -domain adc_op1
-output_supply_port {vout adc_VDD1}
-input_supply_port {vin VDDG}
-control_port {adc_op1_port adc_pwr_ctrl_op}
-on_state {ON vin {adc_op1_port}}
-off_state {OFF {!adc_op1_port}}
```

Upf model 3 Upf for the top-level of architecture

```
####Upf script for top level

###a2d internal module and output module
####Loading the upf for submodules
set_scope adc1/adc1
load_upf "path of internal module upf"
set_scope
set_scope adc1/output_reg1
load_upf "path of output module upf"
set_scope
###Default power domain definition
create_power_domain top -include_scope -elements {tmr1 adc1 fifo1}
###Default power domain supply nets
create_supply_net VDDG  -domain top
create_supply_net VSS  -domain top
###Default power domain supply ports
create_supply_port VDDG  -domain top
create_supply_port VSS  -domain top
###connect supply nets to ports
connect_supply_net VDDG -ports VDDG
connect_supply_net VSS -ports VSS
###Primary supply nets of default power domain
set_domain_supply_net top  -primary_power_net VDDG  -primary_ground_net VSS

#############A2D INTERNAL MODULE#####################
###explicit connection to block levels
###Connect primary supply net of chip to block level port
connect_supply_net VDDG -ports {adc1/adc1/VDDG}
connect_supply_net VSS -ports {adc1/adc1/VSS}
###Definition of isolation for A/D internal module
set_isolation ADC_ISO  -domain adc1/adc1/adc_ctrl
-isolation_power_net VDDG  -isolation_ground_net VSS
 -clamp_value 0  -applies_to outputs
###Definition of isolation control logic for A/D internal module
set_isolation_control ADC_ISO -domain adc1/adc1/adc_ctrl
-isolation_signal adc1/adc_pwr_mgr_int/adc_iso
-isolation_sense high  -location parent
#############A2D OUTPUT MODULE#####################
###Similar to internal module
```

Tool commands 1 Synthesis script

```
#Synthesis script

analyze {List of SystemVerilog models for the architecture}
elaborate
load_upf {Path of the top-level upf file}
###Specify voltage levels for power supply nets
set_voltage 1.1 -object_list
 {VDDG adc1/adc1/adc_VDD adc1/output_reg1/adc_VDD1}
set_voltage 0.0 -object_list {VSS}
###specify the clock for synthesis
create_clock -name ...
set_dont_touch_network  [ find clock ...]
create_clock -name ...
set_dont_touch_network  [ find clock ...]
set_dont_touch_network reset
set_clock_uncertainty 10.0 [all_clocks]
set list [all_inputs]
###Specify the timing constraints
set ports [remove_from_collection $list {clock_ports}]
set_input_delay for all ports
set_output_delay for all ports
###Specify operating condition
set_operating_conditions ...
###Check design and perform synthesis
check_design
check_design -multiple_designs
compile -exact_map
###Results
###Gate-level netlist
write -hierarchy -format verilog -output ./arch_model.v
###Timing constraints file
write_sdc ./arch_model.sdc
###Output Upf
save_upf ./arch_model.upf
```

Tool commands 2 Physical synthesis script

```
####Physical synthesis script
###Set milkyway power nets
set mw_logic0_net VSS
set mw_logic1_net VDDG
create_mw_lib ...
set_tlu_plus_files ...
import_designs {Gate-level Verilog model after synthesis}
read_sdc {Timing constraints file after synthesis}
set_operating_conditions ...
initialize_floorplan -control_type aspect_ratio ...
create_fp_placement -congestion_driven
load_upf "path of UPF generated by synthesis tool"
###Voltage areas are defined
create_voltage_area -power_domain adc1/adc1/adc_ctrl
create_voltage_area -power_domain adc1/output_reg1/adc_op1
###Refine the floorplan
shape_fp_blocks
create_fp_placement -incremental all
###Create logic nets for power nets
derive_pg_connection -create_nets
derive_pg_connection -all
###Map physical power switch and place the switch
map_power_switch  -domain adc1/adc1/adc_ctrl ...
create_power_switch_array -lib_cell "adc1/adc1/SW" ...
connect_power_switch -source adc1/adc_pwr_mgr_int/adc_pwr_ctrl ...
###Do similar stuff for output module
legalize_placement
###Reconnect logical connections
derive_pg_connection -reconnect
###Set constraints for every voltage domain
set_fp_rail_voltage_area_constraints -voltage_area adc1/adc1/adc_ctrl ...
####Do similar stuff for output module and default voltage areas
###Synthesize and commit the power plan
synthesize_fp_rail -synthesize_voltage_areas ...
commit_fp_rail
###For all the power nets, perform this step
preroute_standard_cells ...
###Clock tree synthesis and routing
clock_opt -only_cts
route_opt
###Gate-level netlist
write_verilog -pg arch_model_pwr.v
write_verilog arch_model.v
###Outputs extracted from tool
write_sdf arch_model.sdf
save_upf arch_model.upf
write_parasitics
```

Tool commands 3 Simulation script

```
##This is a sample script for simulation

####Post-synthesis simulation
vcs +v2k -sverilog testb_arch.v arch_model.v
 -v {Verilog model from library} -gui

./simv
####Post-layout simulation
vcs +v2k -sverilog testb_arch.v arch_model)pwr.v
 -v {Verilog model from library with power connections} -gui

./simv
```

Tool commands 4 Power analysis script

```
####Power analysis script

set power_enable_analysis TRUE
###set the inputs required for power analysis
set target_library {path of the .db library of technology library}
set link_library {path of the .db library of technology library}
read_db $target_library
read_verilog {gate-level Verilog netlist obtained from physical synthesis}
current_design {name of top level in the architecture Verilog model}
link
set power_analysis_mode time_based
read_parasitics -triplet_type max {path of .spef.max parasitic information}
read_sdf {path of .sdf file from the Synopsys IC Compiler}
load_upf {path of the .upf file from the Synopsys IC Compiler}
read_vcd {path of .vcd output from post-layout synthesis}
 -strip_path testb_arch/AM1 -time {start-time  end-time}
create_clock inst_clock -name inst_clock -period {period of clock}
create_clock port_fifo1_ntwk_clock -name
 port_fifo1_ntwk_clock -period {period of clock}
###perform power analysis
update_power
###Obtain analysis results
report_power -hierarchy >> Report_power.txt
exit
```

Appendix C

Application Graphs

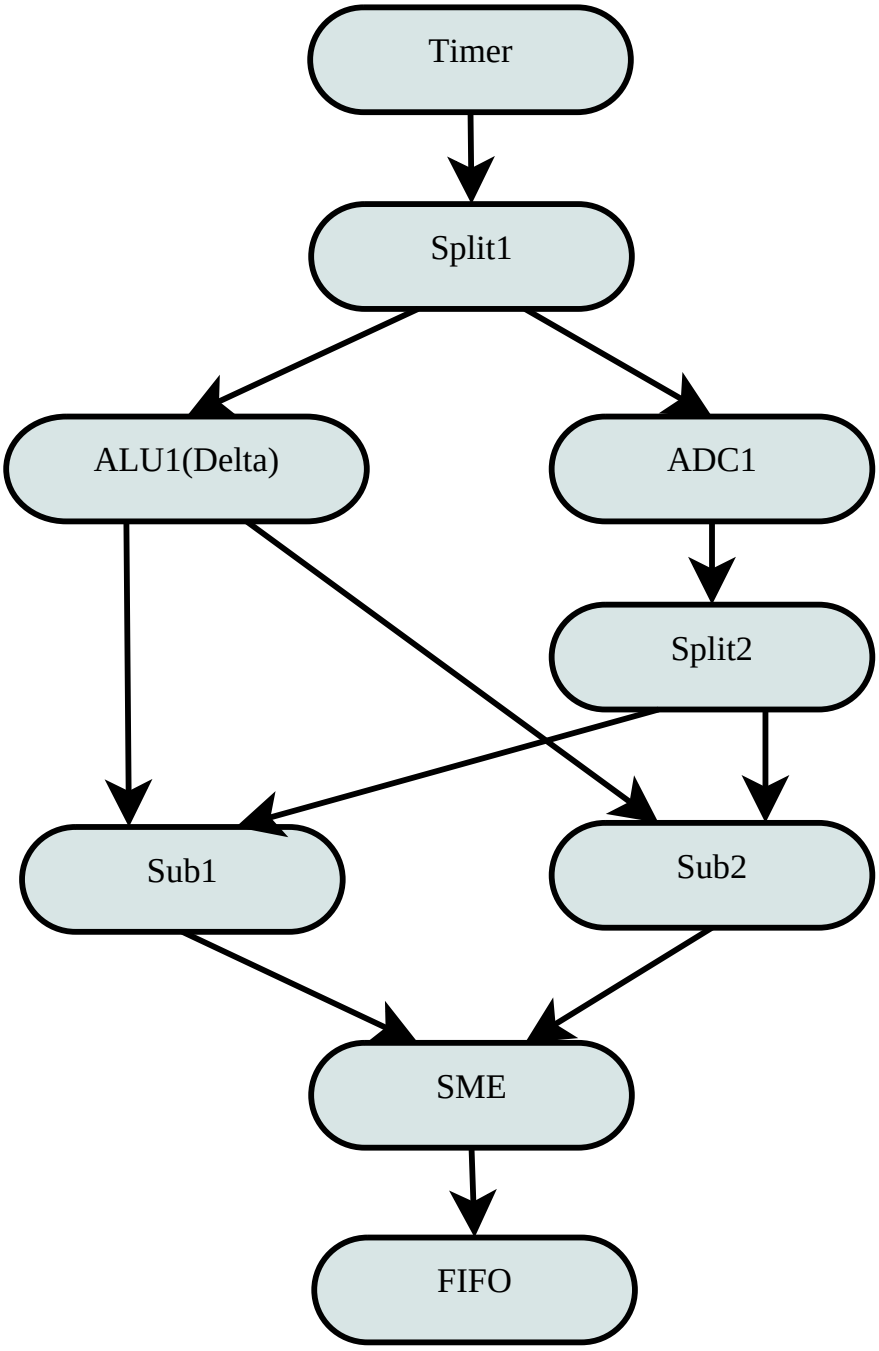

Figure C.1: Application 1

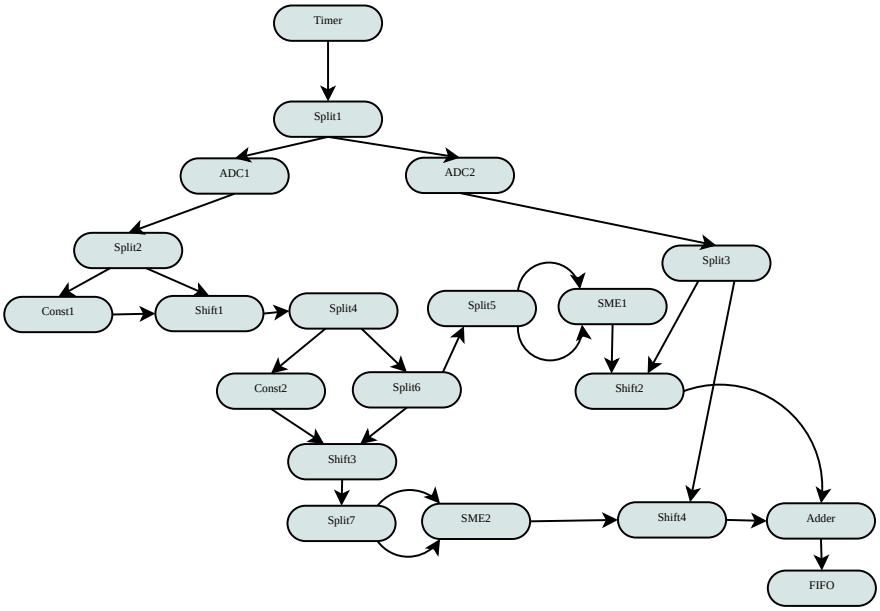

Figure C.2: Application 2

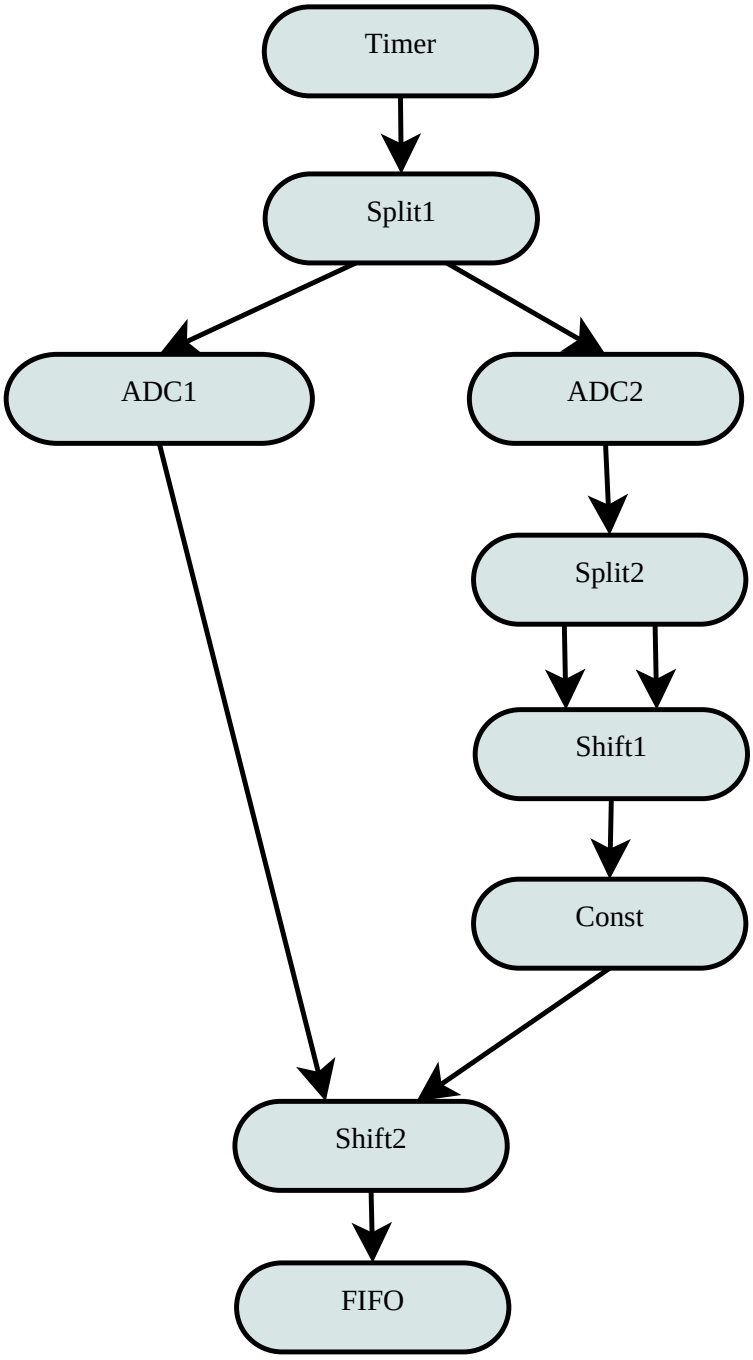

Figure C.3: Application 3

Appendix D

Sample Architectures

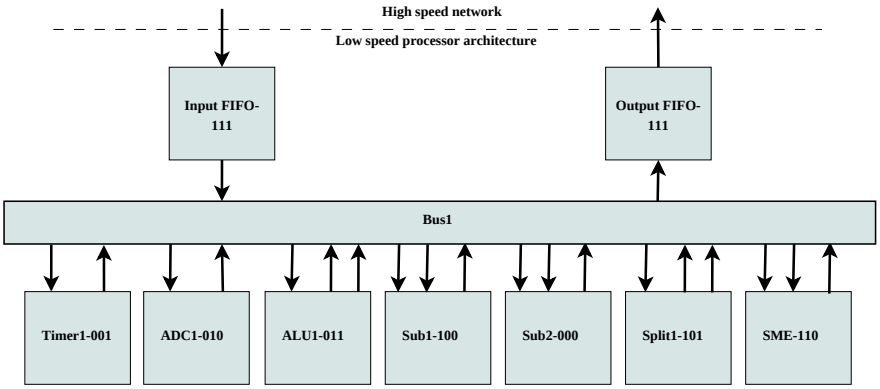

Figure D.1: Architecture 1

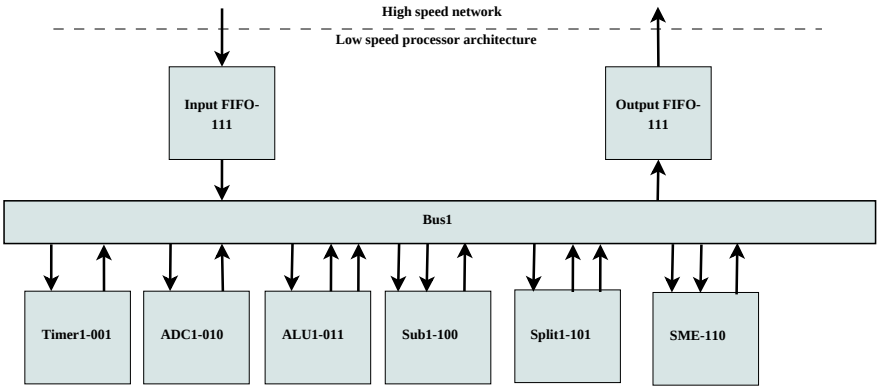

Figure D.2: Architecture 2

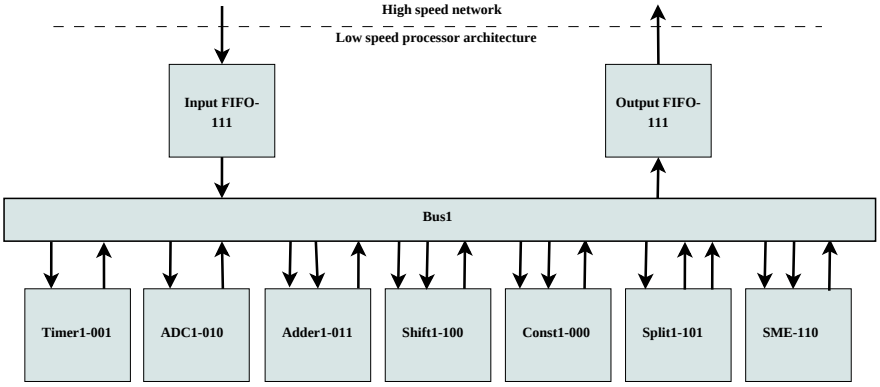

Figure D.3: Architecture 3