DESIGN AUTOMATION FOR LOW POWER RFID TAGS

by

Swapna R. Dontharaju

M.S. Computer Science and Engineering, Pennsylvania State University, 2004

Submitted to the Graduate Faculty of

School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2007

UNIVERSITY OF PITTSBURGH

SCHOOL OF ENGINEERING

This dissertation was presented

by

Swapna R. Dontharaju

It was defended on

July 19, 2007

and approved by

Dr. Alex K. Jones, Assistant Professor, Electrical and Computer Engineering Department

Dr. Marlin H. Mickle, Professor, Electrical and Computer Engineering Department

Dr. James T. Cain, Professor, Electrical and Computer Engineering Department

Dr. Ronald Hoelzeman, Associate Professor, Electrical and Computer Engineering Department

Dr. Bryan Norman, Associate Professor, Industrial Engineering Department

Dissertation Director: Dr. Alex K. Jones, Assistant Professor, Electrical and Computer Engineering Department

DESIGN AUTOMATION FOR LOW POWER RFID TAGS

Swapna R. Dontharaju, PhD

University of Pittsburgh, 2007

Radio Frequency Identification (RFID) tags are small, wireless devices capable of automated item identification, used in a variety of applications including supply chain management, asset management, automatic toll collection (EZ Pass), etc. However, the design of these types of custom systems using the traditional methods can take months for a hardware engineer to develop and debug. In this dissertation, an automated, low-power flow for the design of RFID tags has been developed, implemented and validated. This dissertation presents the RFID Compiler, which permits high-level design entry using a simple description of the desired primitives and their behavior in ANSI-C. The compiler has different back-ends capable of targeting microprocessor-based or custom hardware-based tags. For the hardware-based tag, the back-end automatically converts the user supplied behavior in C to low power synthesizable VHDL optimized for RFID applications. The compiler also integrates a fast, high-level power macromodeling flow, which can be used to generate power estimates within 15% accuracy of industry CAD tools and to optimize the primitives and / or the behaviors, compared to conventional practices. Using the RFID Compiler, the user can develop the entire design in a matter of days or weeks. The compiler has been used to implement standards such as ANSI, ISO 18000-7, 18000-6C and 18185-7. The automatically generated tag designs were validated by targeting microprocessors such as the AD Chips EISC and FPGAs such as Xilinx Spartan 3. The corresponding ASIC implementation is comparable to the conventionally designed commercial tags in terms of the energy and area. Thus, the RFID Compiler permits the design of power efficient, custom RFID tags by a wider audience with a dramatically reduced design cycle.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

I am highly indebted to my advisor, Dr. Alex K. Jones, who provided me the support to carry out my research and made my stay at the University of Pittsburgh productive and enjoyable. He has always made time to talk to me about both research and career issues. Without his guidance and persistent help, this dissertation would not have been possible. I would like to express my deep gratitude for Professors Marlin Mickle and Tom Cain for providing recommendations and suggestions which were invaluable for my research. I also thank my other committee members, Professors Ron Hoelzeman and Bryan Norman for their guidance.

I would like to thank my student colleagues and friends in the EDA lab, the RFID Center for Excellence and at the University of Pittsburgh who have helped me in many ways. I would like to thank the administration and the staff in the Department of Electrical and Computer Engineering who have made it a great experience to be a part of the Department.

Words alone cannot express the thanks I owe my husband, Muthukumar, my mother, Vidya and my sister, Sudha for their unconditional love, patience, understanding and support. Most of all, I am grateful to my father for being my incessant source of inspiration. I dedicate this dissertation to his memory.

1.0 INTRODUCTION

Radio Frequency Identification (RFID) systems are expanding rapidly with their applications in a wide range of areas. RFID systems consist of Radio Frequency (RF) tags and RF readers or interrogators. These systems are used for a wide range of applications that track, monitor, report and manage items as they move between different physical locations. The tags consist of integrated circuits and an RF antenna. A wide range of extensions such as memory, sensors, encryption, and access control can be added to the tag. The interrogators query the tags for information stored on them, which can include items like identification numbers, user written data, or sensory data. Tags contain a unique tag identification number and potentially additional information of interest to manufacturers, healthcare organizations, military organizations, logistics providers and retailers, or others that need to track the physical location of goods or equipment.

RFID tags generally come in two types, active and passive. Active tags require an internal power source to power the tag for receiving queries and transmitting responses. Passive tags do not contain an internal power source and receive energy from the interrogator query. This energy is used to power the tag to determine and send a response to the query. This energy may not be sufficient for intensive computation, limiting the complexity of response. The range of passive tags is also significantly lower compared to active tags. Active tags, on the other hand, are more costly than passive tags and may also require frequent battery replacement.

RFID applications are numerous and far reaching. The most widely used applications can be categorized as those for supply chain management, security, and the tracking of important objects and personnel. Though standards (ISO/IEC JTC1, ANSI, EPC etc) have been developed for RFID hardware, software and data management, these applications have customized requirements. The RFID tag circuits for these are implemented in custom designed chips. Such chips can only be used in specific applications, and therefore, are often called application specific integrated circuits (ASICs).

The design, development, and fabrication of Application Specific ICs is expensive and time consuming. The design process of ASICs requires considerable knowledge in digital logic design, which is very different from application programming in high-level languages such as C. In addition, it is a long and a tedious process that involves designing, synchronizing and synthesizing the digital design. Currently, this process involves months for a hardware engineer to complete. To reduce the design time, effort and cost significantly, it is necessary to develop design automation tools that allow the designers to reduce the time to move from specifications to hardware implementations.

The cost of fabrication of ASICs is very high. Small companies and RFID application programmers are, from a cost standpoint, prohibited from designing their own ASICs. As a result, the design of RFID systems is being done by large companies with state of the art hardware design and fabrication capabilities. These companies also drive the direction of the evolution of RFID systems and, hence, the standards, as technological capability is the key to the development of standards. With the design automation tool, smaller companies or RFID application programmers will be able to achieve customized RFID tag ICs implementations in less time, in a cost effective manner.

The market for RFID tags is characterized by rapidly evolving applications and rather short market windows. A key concept for coping with such requirements is the retargeting of system components for different and/or modified applications or standards. The modified applications can be implemented quickly using the design automation flow.

Power optimization is critical in RFID systems because the power budget is limited in the case of passive tags and battery drain needs to be limited in the case of active tags as frequent replacement of batteries may not be feasible. Thus, the tags generated by the design automation tool must be extremely power efficient.

In this dissertation, an automated, low-power flow for the design of RFID tags has been developed, implemented and validated. This dissertation presents the RFID Compiler, which was developed at Electronic Design Automation Laboratory at the University of Pittsburgh.

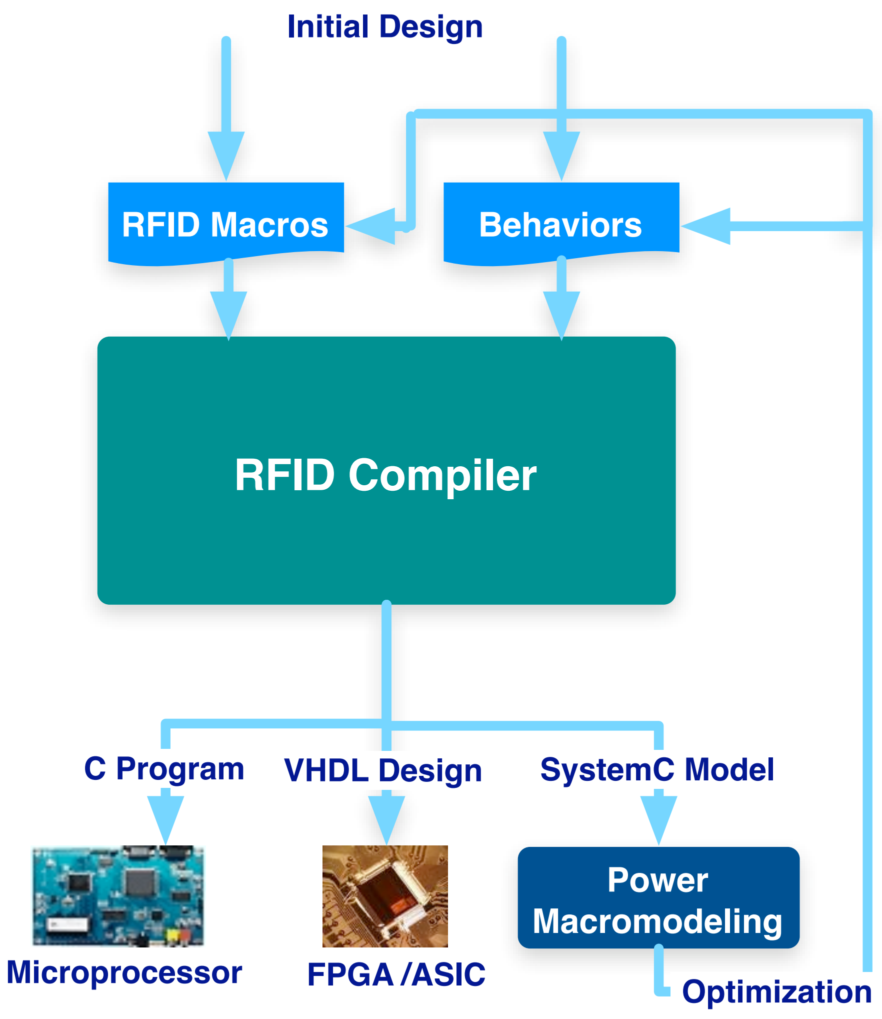


Figure 1: RFID Specification Methodology and Compilation Flow

Figure 1 shows the RFID Compiler flow. The RFID Compiler permits high-level design entry using a simple description of the desired primitives and their behavior in ANSI-C. The compiler has different back-ends capable of targeting microprocessor-based or hardware-based tag controllers. For the hardware-based tag, the back-end automatically converts the user supplied behavior in C to low power synthesizable VHDL. The compiler also integrates a fast, high-level power macromodeling flow, which can be used to generate accurate power estimates and to optimize the primitives and their behaviors.

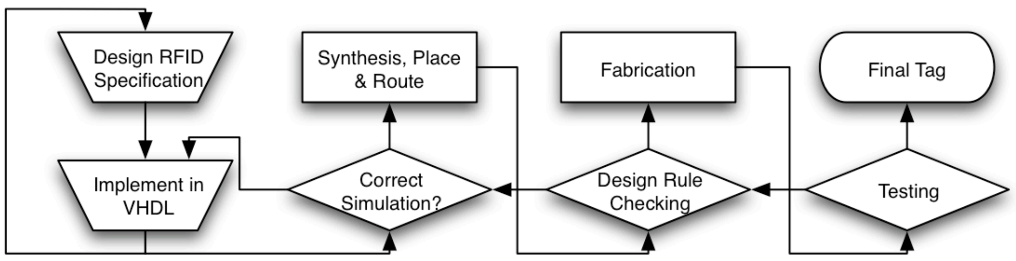
Figure 2 presents a comparison of different RFID tag design methodologies. The current state of the art tag development shown in Figure 2(a) requires lengthy design, fabrication, and testing cycles which can take months, with intellectual property (IP) reuse, to years if developing new IP. A customizable RFID tag, as shown in Figure 2(b), can handle variations in standards and requirements as they are developed with a significantly shorter time to market than current flows. A customizable RFID tag can make flexible RFID systems economically viable. This tag is mass produced and tailored to a particular RFID use after fabrication.

In this dissertation, the design flow from the RFID primitives to the power optimized tag controller is presented. Relative to contemporary technology, the design flow developed and implemented in this dissertation:

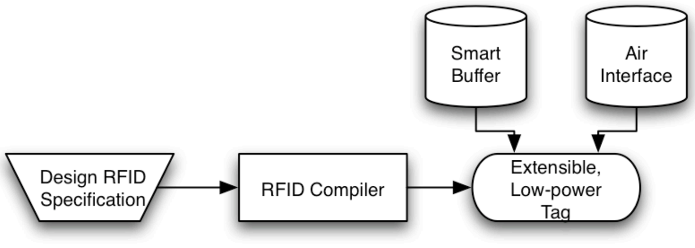
- allows for a simple specification of the RFID primitives and description of the primitives behavior in C
- allows RFID technology design by a wider audience, not necessarily hardware designers
- generates a low power design
- quickly generates accurate power estimates, which can be used for exploration and optimization of both the standard and the C code for behaviors.

This dissertation presents the algorithms, approaches, and techniques used in the design flow of the RFID Compiler. The main emphasis of the mechanisms used in this research is to shorten the design time while producing an implementation which is power efficient.

Several standards have been implemented with the RFID Compiler. Two System-on-a-Chip implementations for the ISO 18000-7 and ANSI standards were first used to validate



(a) Current RFID tag design flow. All tag components integrated manually. Estimated time: months or years.



(b) Automated RFID tag design flow. Prepackaged extensible silicon device. Estimated time: hours or days.

Figure 2: Comparison of RFID tag design philosophies.

the design flow. First, a C program was automatically generated and compiled for the microprocessor-based system. Second, the microprocessor was replaced with a block of low-power FPGA logic. For, the 18000-6C and 18185-7 standards, VHDL was generated, synthesized for ASICs and compared with the corresponding manual implementations. Power estimates were generated using the power macromodeling flow and were compared with the power estimates from the traditional methods to validate the accuracy and the gain in designer productivity.

2.0 CONTRIBUTIONS

There are two major problems in the current RFID systems. (1) Custom tags, with additional capabilities beyond those specified in existing RFID tag standards, are sometimes required for specific applications. To build such custom systems from scratch is generally cost prohibitive and requires long design times. (2) Power optimization is critical in RFID systems because the power budget is limited for passive RFID systems and battery drain needs to be limited for active RFID systems as frequent replacement of batteries is often not feasible after deployment.

2.1 PROBLEM STATEMENT

The objective of this dissertation is to solve the above problems by presenting a standard design flow for the rapid development of RFID tags with custom capabilities, for a wide variety of applications. To address the problem of rapid design of custom tags the RFID specification methodology and compilation flow will automatically generates RFID tag controller code based on a high-level description of the commands to be implemented. The design methodology is proven by targeting embedded microprocessor-based and hardware-based prototypes.

To address the problem of developing low-power tags, this dissertation integrates power as one of the primary metrics early in the design flow. The compiler automatically generates an application specific simulator for the specified design and accurately estimates the power consumed by the design with a factor of 100 speedup over traditional power estimation methods.

2.2 CONTRIBUTIONS

2.2.1 RFID Compiler for the Microprocessor-based Tag

The RFID compiler allows *RFID primitives* or the transactions employed by the RFID systems to be specified using *RFID macros*, an assembly-like format. These RFID macros are processed to generate a template file to specify the behavior for each primitive or macro. All behavior is specified using ANSI-C allowing the user to create arbitrarily complex behaviors. Finally, the RFID compiler generates a C program as output that is compiled onto the microprocessor with its platform-specific C compiler.

2.2.2 Hardware RFID Compiler with VHDL Behavior

While the embedded processor approach does provide a reasonable power/energy consumption, improvement is still possible. To improve both the power and capacity of the controller, a hardware-based RFID controller is explored. The hardware RFID compiler processes the original RFID macros to generate VHDL template files to specify the behavior. The behavior for each primitive is specified using VHDL and the remaining code segment for packet packing, unpacking, and decoding is automatically generated and output in VHDL rather than in C. Finally, the RFID compiler generates the tag controller VHDL design which is synthesized, mapped, placed, and routed for the target hardware using commercially available tools. ASIC implementation of the primitives will be evaluated to reduce the energy.

2.2.3 Hardware RFID Compiler with C Behavior

Because C is a significantly more universally known programming language than VHDL or Verilog, it is desirable to continue allowing the end-user specify the primitive behaviors for the RFID Tag in C. The extended hardware RFID compiler can read primitive behavioral descriptions in ANSI-C and generate synthesizable VHDL for combinational implementation. These combinational blocks are combined with the automatically generated packet packing, unpacking, and decoding VHDL and synthesized for the reprogrammable hardware target.

2.2.3.1 EPC C1 Generation 2 Hardware RFID Compiler EPCglobal Class-1 Generation-2 UHF (Gen-2) specification, recently standardized as ISO 18000 Part 6C, is becoming widely accepted in the supply chain today and is driving the development of passive tags. The communication primitives of Part 6C are significantly different and more complex than the ISO 18000 Part 7 standard. This is because the standard relies on multiple variables and storage of state at several points during each communication operation. This makes the RFID compiler significantly more general than the original implementation. The Hardware RFID Compiler with C behavior is extended with features to support Part 6C primitives and is optimized for low-power and area.

2.2.4 Techniques For Optimizing The Tag Power Consumption

To enable the design of low-power tags, a power macromodeling flow is implemented, which calculates power at a high level during the RFID compiler design automation process. In this flow, the RFID compiler automatically generates a SystemC-based application specific simulator for the input specification. Through access to a pre-profiled library of blocks in the target fabrication process, the power consumption is estimated within an accuracy of 15% of the conventional ASIC power estimation flows, while being 100 times faster. The estimates can be used for the optimization of the primitive behaviors and in the evaluation of alternate protocol designs.

The remainder of the dissertation is organized as follows: Chapter 3 presents the background for this dissertation and the related work. Chapter 4 describes the stages of the RFID specification methodology and the compilation flow in detail for the microprocessor-based system. Chapter 5 describes the Hardware RFID Compilation flow with the behavior specified in VHDL. Chapter 6 describes the Hardware RFID Compilation flow with the behavior specified in C. Chapter 6.4 describes the EPC C1 Generation 2 Hardware RFID Compilation flow. Chapter 7 describes the techniques for optimizing the tag power consumption implemented in the compiler. Finally, the conclusions are presented in Chapter 8.

3.0 BACKGROUND

A considerable body of literature exists on RFID systems. This chapter provides a brief description of RFID systems, their architecture, the tag characteristics, the prevalent standards and common RFID applications.

The focus of this dissertation is to present a standard design flow for the rapid design of low-power RFID tags with custom capabilities for different applications. The automatically generated tag controller code targets embedded microprocessor-based and hardware-based tags. To provide an understanding of the design process and gain in design times, this chapter covers the relevant aspects of the traditional ASIC design flow. The design flows for the reconfigurable field programmable gate arrays (FPGAs) and embedded microprocessors are also included.

This chapter also includes a section describing the commercial RFID systems and other research programs that build customizable RFID systems. Finally the prototyping environment used in the University of Pittsburgh compilation flow is described.

3.1 INTRODUCTION TO RFID SYSTEMS

RFID technology is an alternative to barcode technology and it enables identification at a distance without a line of sight. Figure 3 shows common devices that employ RFID tags. Figure 4 shows an RFID reader that is used to communicate with the tags. Electronic tagging is superior to barcodes in many ways. It allows writing data into the tag, interaction with sensors, scanning a large number of items simultaneously without human error, etc. It supports a much larger set of unique IDs and additional data such as the manufacturer ID.

Figure 3: Common RFID Tags

RFID is not a new technology. For instance, the principles of RFID were employed by the British in World War II to distinguish allied aircrafts from enemy aircrafts. During the 1960s, work on employee access control was carried out at Los Alamos National Laboratories. For many years this technology has been used in applications as diverse as: animal tracking, automatic toll collection and many forms of ID badge for access control. Recently, RFID has become more mainstream. RFID tags can now be achieved at low manufacturing costs and are being adopted in many new applications.

3.1.1 RFID Architecture

An RFID system mainly consists of tags and readers. The reader, also called the interrogator, sends and receives RF data to and from the tag via antennas. The tag, or transponder, is made up of the microchip that stores the data and an antenna. The information collected from the tags is stored in a back-end database. Figure 5 shows the main components of an RFID system.

3.1.2 RFID Tag Characteristics

Tags can vary in terms of the frequency at which they communicate, the protocol, how they are powered and how they store data.

Many types of RFID devices exist, but at the highest level, they can be divided into active and passive devices. Active tags require a power source and use energy stored in a

Figure 4: RFID Reader

battery. The active tag's lifetime, and hence the number of operations, is limited by the stored energy. They have ranges of over a hundred feet. They typically cost more than the passive tags and are used to track high value goods like vehicles and pallets.

In the case of passive tags, the reader is responsible for powering and communicating with the tag. The reader transmits a low power radio signal through its antenna to the tag, which in turn receives it through its own antenna to power the integrated circuit. As a result, passive tags transmit information over shorter distances, typically less than 10 feet. Since they cost considerably less, they are used in tracking low cost items. They do not require batteries and have an indefinite operational life.

3.1.2.1 Communication There are many different versions of RFID systems that operate at different radio frequencies. The choice of frequency is dependent on the requirements of the application. Three primary frequency bands have been allocated for RFID use. The Low Frequency band (125/134KHz), is most commonly used for access control and asset tracking. The High Frequency band (13.56 MHz) is used where medium data rate and read ranges are required. The Ultra High Frequency (850 MHz to 950 MHz and 2.4 GHz to 2.5 GHz) band offers the longest read ranges and highest reading speeds. The techniques in this dissertation can be applied regardless of the frequency employed.

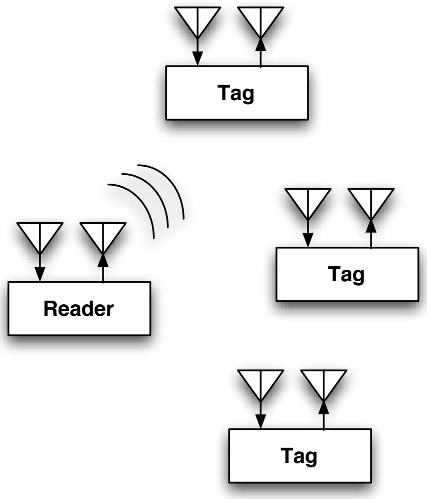


Figure 5: The Main Components of an RFID System

3.1.2.2 Computation Most passive tags have simple or no computational capabilities. They may only have a simple memory that can be remotely accessed. They may have a simple design that can perform certain XOR and AND operations. Active tags can be more complex, and could have have a few thousand logical gates to implement logic.

The two basic types of memory available on RFID tags are read-only memories and read-write memories. Read only chips are programmed with unique information stored on them during the chip manufacturing process. The information on read only chips can never be changed. With read write memories, the user can add information to the tag or write over existing information when the tag is within range of the reader.

3.1.2.3 Security Applications for RFID continue to expand into domains such as electronic passports, electronic payment systems, and electronic container seals. These applications have a risk of unauthorized access to sensitive biometric or financial information through the RFID tag or or tag communication. RFID devices are susceptible to many forms attacks, which may affect the security and privacy of the individual users or the organizations. Some of the main attacks possible are physical attacks against tags, counterfeiting of tags or readers, eavesdropping messages transmitted in protocols, etc. Some of the attacks that are less malicious are traffic analysis to detect when and how many messages are sent, disrupting messages and denial of service attacks.

However, as RFID devices are intended to be small and relatively simple devices, security protocols and techniques can significantly lag behind the other details such as correctness, read rate, power consumption, etc. This is primarily because strong authentication and encryption algorithms are complex and would significantly increase the cost and power budget of the tag. As such, the state of security in RFID systems is generally weak compared to other mature computational technologies such as Internet servers, shared computing workstations, and even smart-cards. Various security techniques have been proposed in the recent years, including killing tags at the checkout point, physical tag memory separation, rolling codes, hash locks, challenge-response protocols, etc. An exhaustive study of the security attacks of RFID as well as the protection techniques is beyond the scope of this dissertation, but can be found in [5]. Design automation flow discussed in this dissertation could allow the RFID primitives to support novel security techniques.

3.1.3 RFID Applications

RFID is expected to provide huge advantages to manufacturers by offering the tools to better plan production and respond more quickly to market demand. The use of RFID tags will permit automatic management of stock and inventories in shops and warehouses. Supermarkets and other retailers across the world are pioneering large-scale item-level deployments of RFID in consumer goods. Some examples are: Wal-Mart in the US, Marks & Spencer and Tesco in the UK and Metro in Germany. By using RFID technology for tracking sales, stock and orders they aim to lower operational costs which in turns impacts the pricing.

The United States Department of Defense has been using active tags to reduce logistics costs and to improve supply chain visibility for more than 15 years. The US government is considering the use of RFID tags in the passports to reduce counterfeiting and to enable automatic identity checking. The European Union is planning to incorporate RFID tags in the European paper currency to make forgeries difficult and to provide tracking of its use.

Management of books can be automated by using RFID for libraries. Tags can be inserted in each volume, thus simplifying the work of library staff as well as improving the experience of users. Inventories can be carried out without removing books from the shelves, by automatically detecting missing or misfiled books, or even by using an automatic sorter for the returned volumes. An example is the use RFID tags in the Vatican Library in Rome to identify and manage its extensive book and document collection [6].

Many access control applications also employ RFID tags. The use of RFID cards or badges makes access control simpler for people as they do not have to manipulate their identification card. It can also be used for identifying people or for safety reasons, e.g., when a building must be evacuated. Theft and vandalism are also impeded. Another example of access control is the keyless and passive entry systems in cars. The owner initiates a secure exchange of information between the car's remote unit and the car by the push of a button on the remote, and the door of the car automatically unlocks itself.

RFID technology is very useful in location sensing, item/animal tracking, healthcare, etc. Examples include a location sensing prototype system for locating objects inside buildings [7] and a system for identifying persons and objects inside and outside hospitals [8]. RFID tags and intelligent transponders are widespread for vehicle to roadside communications, road tolling and vehicle access control. Different types of RFID systems are being developed to support all aspects of aviation baggage tracking, sorting and reconciliation [9].

This motivates the need for different RFID customizations and possible interoperability across domains. This can be easily accomplished using the concepts in this dissertation.

3.1.4 RFID Standards

There are number of standards for RFID systems which have either been published or are in the process of being elaborated. The main specifications are: ISO (International Organization for Standardization [10]) standards, EPC (Electronic Product Code [4]) specifications and ANSI (American National Standards Institute [11]).

The key active ISO standard is 18000-7 [10], which is an international standard that defines the air interface for RFID devices used in item management applications. The standard defines the forward and return link parameters for an active RFID air interface at 433 MHz and the communications protocol used. The ISO 18185 Part 1 standard [12] is an international standard that provides a system for the unique identification and presentation of information about freight container electronic seals. It is used in conjunction with the other parts of ISO 18185 such as, Part 4 that specifies data protection and Part 7 that specifies the physical layer protocol.

The EPC specifications, established by EPCGlobal, distinguish several classes of tags according to their function. Class 1 corresponds to the most simple tags, which have only a unique identifier for the tag by default. Class 2 offers more memory and allows authentication functions to be carried out. Class 3 corresponds to semi-passive tags and finally class 4 corresponds to active tags, which can potentially communicate with each other. Of these, the Class 1 Generation 2 (or "Gen 2") UHF specification is the most widely used [4]. It has been integrated with the ISO standards as the 18000-6C passive standard [13].

The ANSI/NCITS 256-2001 is the American National Standard for RFID devices [11]. It is intended to allow for compatibility and to encourage interoperability of products for the growing RFID market in the United States.

As part of this dissertation, I show the comparison of automated implementations of these standards and in some cases, merge the implementations of different standards such as ISO 18000-7 and ANSI/NCITS 256-2001.

3.2 HARDWARE DESIGN METHODOLOGIES

Historically, digital hardware has been divided into two main groups, general-purpose processor and application specific hardware. A general-purpose processor is a fixed architecture device which implements a pre-defined set of instructions. General-purpose processors can be classified as: microprocessors and digital signal processing (DSP) processors among others. Examples of microprocessors are Intel's Pentium family, Sun's UltraSparc family, Intel's XScale, and IBM's PowerPC for embedded applications, etc. These processors execute programs stored in some internal or external memories by fetching their instructions, examining them and then executing them one after another. An organization of a simple bus-oriented microprocessor is shown in Figure 6. New programs can be loaded into memory as needed. The computation of any algorithm is determined by the software program, not the hardware. Because their instruction sets include very general applications such as arithmetic and logical operators, general-purpose processors can be programmed to perform any conceivable function. The application can be programmed in a high-level language such as C and is compiled for execution on any processor. However, general-purpose processors are slower than dedicated hardware at performing computationally intensive functions. They also consume higher power compared to application specific hardware.

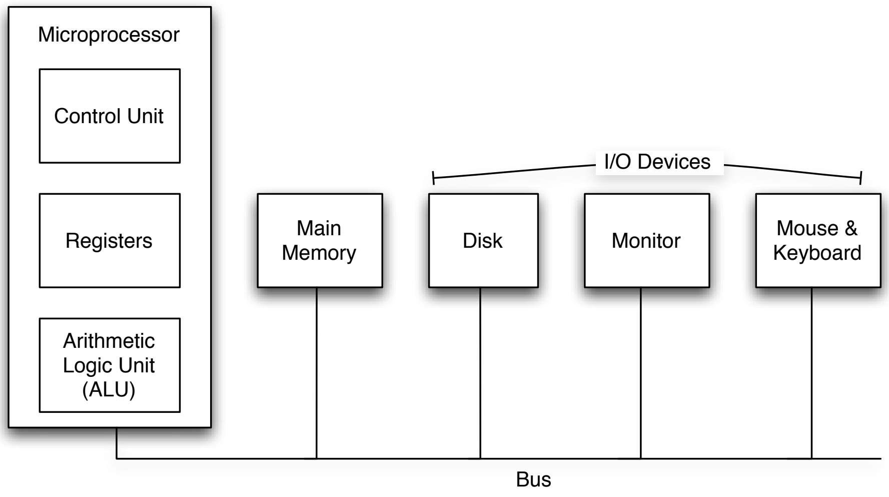


Figure 6: Organization of a Simple Bus-oriented Microprocessor

For application-specific hardware, an engineer designs all of the circuits specifically for an application. These circuits or custom hardware implementations, which are often referred to as application-specific integrated circuits (ASICs), usually lead to better performance than general-purpose processors since they can be optimized for the specific application. However, if a new function is required, then an entirely new ASIC must be created. Another disadvantage of ASICs is the labor-intensive design cycle. It typically takes months or years for hardware engineers to design a new ASIC and have it fabricated and tested. This also translates into a high cost and long time-to-market. In spite of these drawbacks, application specific hardware is widely used whenever performance is of primary importance or product volume is very high. By optimizing the hardware for a particular task, an ASIC can often achieve computation speeds several orders of magnitude faster than doing the same computation using general-purpose hardware, while also requiring lower power. ASICs shine in extremely high volumes of production of relatively unchanging specification as they provide the best performance and power solution.

In recent years, programmable logic devices have been increasingly gaining interest, when high volume is not possible, but custom hardware is desired. They have some of the advantages of both general-purpose and application-specific hardware. These devices are most commonly commercially available Field Programmable Gate Arrays (FPGAs) or Complex Programmable Logic Devices (CPLDs). While FPGAs are large, have higher performance and consume higher power, CPLDs are small, have lower performance and consume lower power. These devices provide a relatively large number of programmable functional units and programmable interconnections. The functionality of the hardware is determined by how the functional units and interconnections are configured. By changing the configuration, the hardware can be made to perform a completely different function. Different types of applications can be implemented at speeds between application specific hardware and general-purpose processors. In addition, the configuration can be changed relatively quickly from one function to another, giving some of the same flexibility as general-purpose processors. These programmable logic devices consume lower power compared to general-purpose processors and higher power compared to application-specific hardware.

3.2.1 Design Flows For ASICs and FPGAs

The traditional design flow for ASICs is depicted in Figure 2(a). It starts with the development of a hardware definition for the application. This is usually done with a hardware description language (HDL). The two main HDLs in use today are VHDL and Verilog. For HDL coding, a sound knowledge of the digital logic design is required. The functionality of the HDL is then verified in simulation against the initial specification. The HDL description of the design is then synthesized into a netlist consisting of cells and their interconnections. The cells used in the netlist are obtained from a standard cell technology library provided by the ASIC manufacturer, typically analogous to basic logic gates. The library defines the delay models, models for variations of temperature, voltage and manufacturing processes as well as the functionality of each cell. This gate-level netlist is then simulated to verify the functionality of the design. The logic cells are then placed on the layout of the chip as a minimum area arrangement that meets the performance constraints. After placing the logic gates, the interconnections between logic gates are routed according to the specified netlist. Post-layout simulation is performed at this level, mainly to verify that the design meets the specified timing constraints. Pin assignment is then performed to connect the input and output signals of the design to the I/O pins of a chosen frame. Finally, the physical layout of the design is sent off for fabrication.

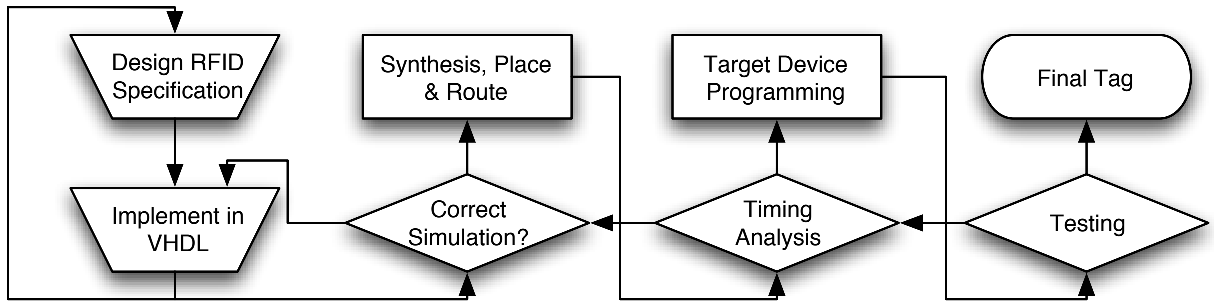


Figure 7: Design Flow For Reconfigurable Hardware

The traditional design flow for reconfigurable hardware is depicted in Figure 7. To map an application to reconfigurable hardware, the designer must first define the hardware structure for the application using a HDL. After that, the HDL code is verified to make sure that it matches the required functionality, prior to synthesis. The design is then synthesized into a technology-dependent netlist. This netlist is specified in terms of the basic logic block of the device. For example, if the Xilinx Spartan-3 series FPGAs are used, the netlist is specified in terms of Configurable Logic Block (CLB). The incoming netlists and constraints are mapped into the available resources on the target device. Then the design is placed and routed onto the device to meet the timing constraints. The timing of the design is then verified by static timing analysis. The placement and routing processes produce the physical implementation for the design, which is then translated into a bit stream (commonly known as configuration file), which is used to program the target device. This flow is considerably simpler and shorter than the ASIC design flow. Moreover, once the functionally correct and synthesizable VHDL is available, mapping to the target device can be done quickly and efficiently using commercial back-end tools.

The compilation flow developed in this dissertation automatically generates the tag C program or synthesizable tag VHDL design for compiling on to microprocessors or for mapping on to programmable logic devices. This avoids the high initial cost, the lengthy development cycles, and the inherent inflexibility of conventional ASICs. RFID tags with custom capabilities for different applications can be designed and prototyped in a short amount of time. However, if the tag design is to be manufactured on a very large scale, ASIC fabrication can be done after prototyping the tag design using a reconfigurable device.

3.3 RELATED WORK

Several RFID tags are being developed in the industry and in research labs. In this section, some of these tags are highlighted. The available customizable RFID systems are also presented. The differences between these approaches and this dissertation are also discussed.

3.3.1 Commercial Systems

Some of the companies that develop RFID tags are Savi, Intermec, Phillips, Motorola, Hitachi, etc. The tags are designed using a traditional ASIC design flow. The SaviTag ST-602 is a simple, low cost, tag for real-time tracking of containers and their contents within facilities or across geographies. It has a battery life of four years, a range of about 300 feet and memory capacity of 36 bytes (see Figure 8(a)) [14]. The SaviTag ST-654 high performance tag is suited for various applications including tracking of shipping containers, vehicles, and other large assets. It is claimed to have immunity to some effects such as dirt and enclosures,

(a) Savi Tag ST-602 (b) Savi Tag ST-654 (c) Savi Reader SR-650

Figure 8: Some RFID Products From Savi [1].

and has a range of up to 300 feet, a typical battery life of five years using lithium cells and memory capacity of 128K (see Figure 8(b)). The Savi tags are compatible with the Savi reader products (for example, see Figure 8(c)).

Other custom tags, which are commercially available, are also designed using the standard ASIC design flow. A novel design of a batteryless, self-powered RFID transponder is presented in [15]. Data transmission uses frequency shift keying (FSK) modulation and the circuit is designed such that the output frequencies are implicitly determined, independent of the load of the antenna. The design is fabricated as a 0.8um CMOS circuit. An integrated circuit for a battery-less transponder system for high performance identification systems is described [16]. The operating principle of the system gives a superior performance in reading distance due to separation of the powering and data transmission phases. The design of a read/write tag targeted towards low-cost applications is described in [17]. An ultra small RFID micro-chip storing a unique 128-bits ROM ID code, for use in a reliable authentication through a network-based secure ID management is available [18]. An ultra-small radio-frequency identification chip, called the u-chip, has been developed for use in a wide range of individual recognition applications [19] . The chip is fabricated using 0.18um standard CMOS technology. The RFID enabled micro-chip [20] is small sized and low cost, and is suitable for attachment to paper media and small products, aiding counterfeit prevention and product tracking in market environments. A wireless sensor prototype platform (UbiSensor) presented in [21] combines sensors and RFID resulting in sensing, data processing, network protocol execution, and energy scavenging capabilities. The platform design is driven by energy consumption minimization of given tasks. A commercially available microcontroller, low power RF transceiver, and power generator circuits are used.

3.3.2 Customizable Systems

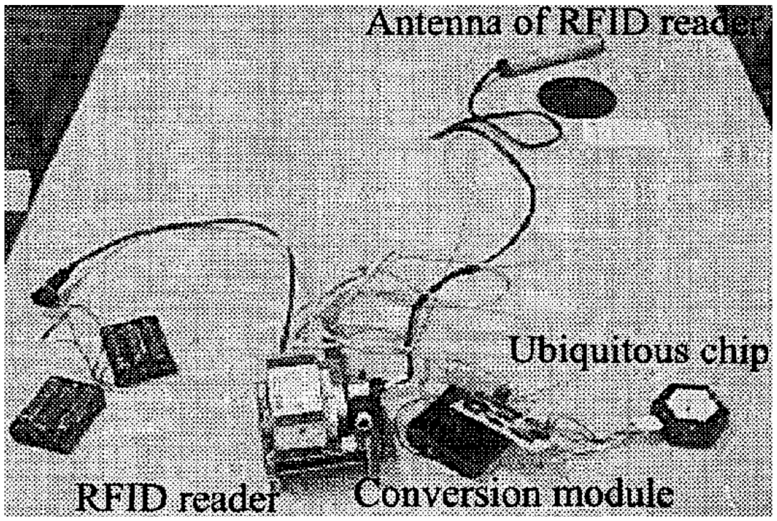


Figure 9: Rule-based RFID tag system. Source: [2].

A rule-based RFID tag system using ubiquitous chips is proposed in [2] to construct flexible and scalable systems. Ubiquitous chips are rule-based I/O control devices, to which several devices such as switches, sensors, LEDs (Light Emitting Diode), etc., can be attached. Ubiquitous chips use ECA rules for event-driven programming. An ECA rule consists of the following three parts: events (E), conditions for executing the actions (C) and the actions to be carried out (A). ECA rules have been used to describe the behaviors of active databases. An active database is a database system that carries out prescribed actions in response to a generated event inside / outside of the database. A simple example of an ECA rule is shown in Figure 10.

```
E: A person passes the entrance.

C: He/She has a license to enter the office.

A: Open the door.
```

Figure 10: Example ECA rule for an employee access control application. Source: [2].

Figure 9 shows a prototype of the system proposed in [2]. An RFID reader and RFID tags made by Texas Instruments (TI-K2A-001A) are used. The reader is connected to a ubiquitous chip through a conversion module. The conversion module converts the ID received from the RFID reader into the ubiquitous chip format. The application program is input into the ubiquitous chips using rules to describe behaviors. Using this system, an employee access control application is implemented. The authors then show how this system can be customized using a different set of rules when the application is extended to support a different type of access control. This approach may be useful when applications are simply extended, however, when a completely new application or a new protocol needs to be implemented the tag itself will require customization. Our RFID design automation flow allows full customization of RFID tags.

One of the main components of an RFID system communication is the physical layer protocol employed to encode bits of information. The physical layer features for the bit encoding mechanism vary across various RFID standards. For example, the ISO 18000 Part 7 active tag standard specifies *Manchester encoding* [22] to transmit encoded data RFID interrogators and tags [10] while the ISO 18000 Part 6C standard defines different physical layer features of transactions among readers and tags. *Pulse-Interval Encoding (PIE)* [13] is utilized to encode data transmitted from readers to tags and either *FM0* [23] or *Miller encoding* [24] is utilized to encode the backscattered data from tags back to readers [13]. [3] describes a methodology by which the physical layer decoder and encoder hardware blocks can be automatically generated from a high-level specification of the protocol. This design flow is shown in Figure 11. The user describes the waveform features of the encoding scheme such as edge transitions, level detection, pulse width detection, etc. from a physical layer specification. The user can then combine one or more wave features to represent bits or groups of bits. The physical layer synthesis tool then automatically generates hardware blocks for encoding and decoding the signal in VHDL. These VHDL descriptions are created from the combination of predefined parameterized hardware libraries and automatically generated hardware blocks for detecting and generating the waveform features in the encoding.

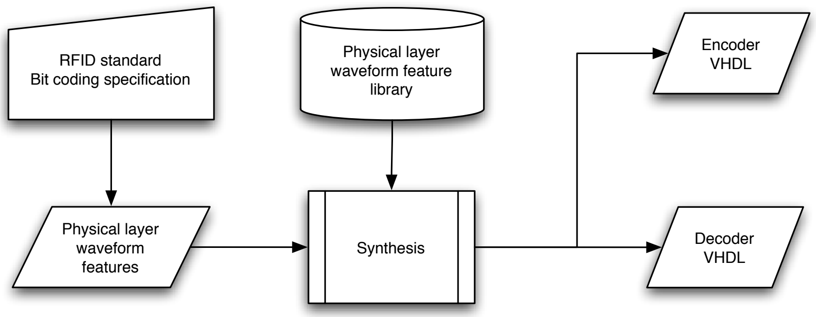


Figure 11: The generation flow for an RFID data encoder and decoder. Source: [3].

3.4 RFID PROTOTYPING ENVIRONMENT

The RFID compiler uses a simple specification of the RFID design to create the RFID tag controller. The complete tag prototype consists of a programmable controller, an air interface, and a power-aware *smart buffer* that sits in between, as shown in Figure 12. The *smart buffer* [25], implemented in an FPGA, contains a small amount of logic to detect whether incoming packets are intended for the tag, thereby allowing the controller to remain powered down to reduce overall system power consumption. The *Air Interface* serves as an interface between the smart buffer and the interrogator, with the necessary receiver and transmitter circuitry to allow the RFID tag to communicate with the RFID interrogator.

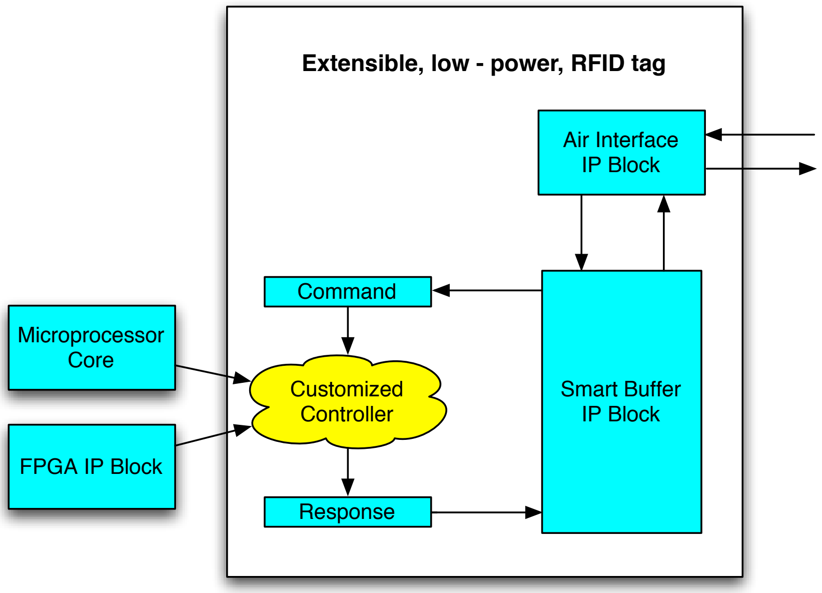


Figure 12: Extensible, low-power RFID tag.

4.0 RFID SPECIFICATION METHODOLOGY AND COMPILATION FLOW

The RFID specification methodology and compilation flow are illustrated in Figure 13. The *RFID primitives* from the specification of the standards and the proprietary extensions are first converted into a simple assembly-like description or *RFID macros*. The first stage of the compiler, the RFID Parser *(rfpp)*, reads this and builds it into the compiler. The user then defines the tag behavior in response to each *RFID primitive* in ANSI-C. To simplify the user interaction, the RFID Parser generates C code templates automatically. The user uses simple ANSI-C constructs to plug in the behavior into the template. The RFID Compiler *(rfcc)* generates the tag controller C code based on the input *RFID macros* and the tag behavior. The C code is compiled using an embedded compiler to generate executable binary for the microprocessor of the tag.

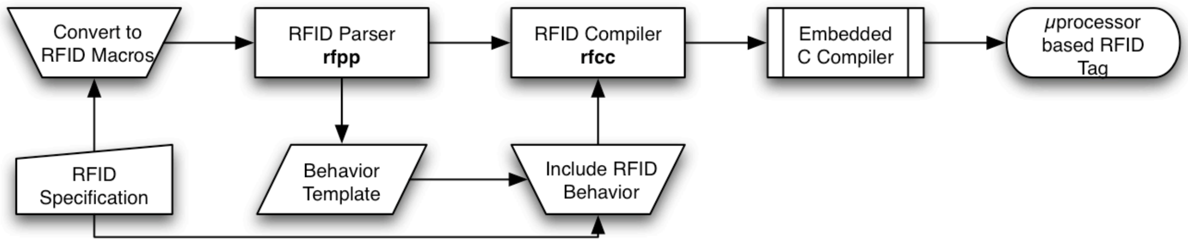


Figure 13: RFID Specification Methodology and Compilation Flow

To provide the background for understanding the inputs to the RFID specification methodology and compilation flow, the basic RFID command structures from the ISO 18000 Part 7 and ANSI-256 specifications are described in Section 4.1. The process for converting the basic RFID structures from the standard specification into a form read by the compiler is described in Section 4.2. The process for inserting the RFID tag response behavior described in the standard into the automatically generated behavior template into the form read by the compiler is shown in Section 4.3. Section 4.4 describes the process to generate the final program to be executed on the tag. Section 4.5 describes the prototype microprocessor based system developed, and the experimental results are shown in Section 4.6.

4.1 DESCRIPTION OF THE PRIMITIVES

4.1.1 ISO 18000 Part 7

As an illustration of the RFID standards specification, a *primitive*, *Collection* command has been selected from the ISO/IEC 18000-7:2004(E) standard. The format of the fields in the interrogator to tag command format for the *Collection* command primitive and its response are shown in Figure 14. Each RFID primitive has a unique field called the `command code` or `opcode`, which serves as the identifier and signals the tag what type of command is being issued. In addition to the `opcode`, each RFID primitive contains a number of other fields of varying lengths as positions for data present as can be inferred from Figure 14. The command contains a `CRC` to ensure the command packet is properly formed. The remainder of the packet contains particular fields appropriate to the command. For example, the command type field indicates the presence of `Tag ID` and `Owner ID` fields. Broadcast commands do not contain a `Tag ID` while point to point commands contain a specific `Tag ID` of the target tag. The `Owner ID` field, which is programmed in the tag's memory, allows the segregation of different groups of tags within the whole population.

Similarly, the tag response includes the `command code`, `CRC` and other data fields. The tag response also includes a `tag status` field, which consists of nested fields such as `acknowledge`, `tag type`, `battery`, etc.

Collection Command

Prefix	Type	Owner Id	Interrogator Id	Opode	Size	Reserved	CRC
8 bits	8 bits	24 bits	16 bits	8 bits	16 bits	8 bits	16 bits

Response

Tag Status	Message Length	Interrogator Id	Tag Id	Opode	CRC
4 bits	1 bit	2 bits	1 bit	2 bits	2 bits

Tag Status

Modefield	Reserved	Ack	Reserved	Tag Type	Reserved	User Id	Battery
4 bits	1 bit	2 bits	1 bit	2 bits	2 bits	1 bit	4 bits

Figure 14: *Collection* Command and Response Format (ISO 18000-7)

4.1.2 ANSI NCITS 256-2001

Figure 15 shows the interrogator to tag command format for the *Get Tag Status* command. As in the case of ISO 18000-7 commands, the command contains a `command code` to signal the tag what type of command is being issued, a `CRC` to ensure the command packet is properly formed and other appropriate data fields. Similarly, the tag response includes the `command code`, `CRC`, and a `tag status` field, which consists of nested fields such as `reserved`, `beeper`, `battery`, etc.

Get Tag Status Command

Command Code	Interrogator Id	Tag Id	CRC
8 bits	16 bits	24 bits	16 bits

Response

Tag Id	Interrogator Id	Tag Status	CRC
24 bits	16 bits	8 bits	16 bits

Tag Status

Reserved	Beeper	Checksum	Battery	Error
4 bits	1 bit	1 bit	1 bit	1 bit

Figure 15: *Get Tag Status* Primitive and Response Format (ANSI)

4.2 MACROS SPECIFICATION

The simple assembly-like descriptions corresponding to the *RFID primitives* and their responses are termed *RFID macros*. Each *RFID macro* description has a short character string that corresponds to the name of the *primitive*, a number corresponding to the value of the *opcode*, a set of operands corresponding to the *primitive's* format and a set of operands corresponding to the response format.

Figure 16 shows the *RFID macros* file corresponding to the Owner ID Write *primitive*. In order to capture the details of the lengths of each field in the *primitive*, the macros file has been conceptually broken into a *declarations* section and a *main* section. The *declarations* section allows the user to pre-declare the lengths of all the fields that will occur in the *primitives* and the responses. This eliminates the need to specify the field's length multiple times as the field can occur in multiple *primitives* and / or multiple responses. In the *main* section, the *primitives* and the corresponding responses are defined in terms of the fields thereof.

In some cases, the fields in the *primitive* or the response have multiple nested fields of varying lengths. These fields can be described with ease, as shown in Figure 16, thereby providing the user with the capability to adopt any level of granularity in manipulating the *primitives* and / or responses. In the *macros* shown in Figure 16, the string used to denote the *owner ID write* command is `ionw`. The decimal value of the command code corresponding to the *owner ID write command* is "137". Figure 17 shows an example *RFID macros* file containing the *Get Tag Status* primitive shown in Figure 15.

The grammar for the RFID specification is shown in Figure 18. The RFID parser parses the macros specification file. The operand declarations and RFID macros are stored in a symbol table. Each macro data structure has the name, opcode, a pointer to the list of command operands and a pointer to the list of response operands.

```
declarations
prefix(8)
type(8)
ownerid(24)
interid(16)
tagid(32)
comcode(8)
siz(16)
res(8)
crc(16)
tagstatus(16)[
   modefield(4)
   reserved1(3)
   acknowledge(4)
   reserved2(2)
   tagtype(3)
   reserved3(1)
   userid(1)
   battery(1)
   ]
mesglen(8)

main
icol(16)   prefix    type    ownerid  interid comcode  siz   res   crc
           tagstatus mesglen interid  tagid   ownerid  crc
ionw(137)  prefix    type    ownerid  tagid   interid  comcode  crc
           tagstatus mesglen interid  tagid   comcode  crc
```

Figure 16: Macros specification (ISO 18000-7).

```
declarations
interid(16)
tagid(32)
comcode(8)
crc(16)
astatus(8) [
reserved(4)
beeper(1)
checksum(1)
lowbattery(1)
error(1)
]

main
asta(23)   comcode   interid   tagid
           tagid     interid   astatus
```

Figure 17: Macros specification (ANSI-256).

```
rfidspec      : declarations macros
macros        : macro
              | macros macro
macro         : name opcode cmdoperands rspoperands
cmdoperands   : operand
              | cmdoperands operand
rspoperands   : operand
              | rspoperands operand
declarations  : nesteddec
              | simpledec
              | declarations nesteddec
              | declarations simpledec
nesteddec     : simpledec LBR simpledecs RBR
simpledecs    : simpledec
              | simpledecs simpledec
simpledec     : string LBR precision RBR
opcode        : integer
precision     : integer
operand       : string
name          : string
```

Figure 18: RFID Specification Grammar.

```
for (each macro in macros list)
{
    get macro name, its response list
    for each (operand in response list)
    {
        get its information from symbol table
        print to template file
    }
}
```

Figure 19: Pseudo-code for Template Generation.

4.3 TEMPLATE FOR BEHAVIOR

The RFID interrogator (Reader) transmits an *RFID primitive* to the tag through an air interface. The tag responds to the interrogator's *primitive* by way of changing its current state and / or transmitting a response message back to the interrogator. The nature of the tag behavior is specified to the RFID compiler, to be incorporated in the end tag software. The user specifies tag behavior in a programming language such as ANSI-C.

To simplify the user interaction, the RFID parser generates a template for the response behavior indicating where the user must specify such custom behavior. Any C language constructs (conditionals, loops, etc.) can be added (or left unchanged) by the user to check the values of the fields of the incoming *RFID primitive* and to specify the values of the fields of the response. The pseudo-code for template generation is shown in Figure 19. The template generated for the *Collection* command (*icol* in the *macros* specification in Figure 14) is shown in Figure 20. A file containing similar templates for all the macros that were included in the macros specification file will be generated for the user.

Figure 21 shows the completed behavior for the *Collection* command. The values of the fields in the response are manipulated as per the ISO 18000-7 specification. All details regarding the size and the position of the fields in the interrogator command and in the response packet are handled automatically by the compiler and need not be specified in the

```
TAGSTATUS.modefield =
TAGSTATUS.reserved1 =
TAGSTATUS.acknowledge =
TAGSTATUS.reserved2 =
TAGSTATUS.tagtype =
TAGSTATUS.reserved3 =
TAGSTATUS.userid =
TAGSTATUS.battery =
RESPONSE.mesglen =
RESPONSE.interid =
RESPONSE.tagid =
RESPONSE.ownerid =
RESPONSE.crc =
```

Figure 20: Template Generated for Behavior of *Collection* Command.

```
TAGSTATUS.modefield = 0;
TAGSTATUS.reserved1 = 7;
if (commandValid)
    TAGSTATUS.acknowledge = 0;
else
    TAGSTATUS.acknowledge = 1;
TAGSTATUS.reserved2 = 3;
TAGSTATUS.tagtype = 2;
TAGSTATUS.reserved3 = 1;
TAGSTATUS.userid = 1;
TAGSTATUS.battery = 0;
RESPONSE.mesglen = 112;
RESPONSE.interid = interid;
RESPONSE.tagid = tagid;
RESPONSE.ownerid = ownerid;
RESPONSE.crc = crc;
```

Figure 21: Behavior of *Collection* Command After User Input.

behavior. Hence the complexities of unpacking the command and packing the response are abstracted away from the user. However, the user's option to manipulate each individual field in the response has been preserved. Thus, the customization of responses and state changes can increase in complexity with user familiarity.

The completed behavior for the *Collection* command command is shown in Figure 21, illustrating how simple C constructs can be used to plug in the response behavior of the tag.

4.4 MICROPROCESSOR-BASED CONTROLLER

The final phase of the compiler is the computer (target microprocessor) code generation based on the input macros specification and the tag behavior. The compiler generates decode instructions that identify the incoming *RFID primitive*. For each case of an incoming command, the compiler also creates routines that unpack the command into the fields that it is expected to contain. The corresponding behavior is then attached to each case of an incoming command. The routines for packing the response are then generated. The result is that the final generated RFID C program, from the above steps, receives the incoming *RFID primitive*, identifies it based on the value of its opcode, unpacks its fields, executes its behavior, packs its response and sends it to the interrogator.

Figure 22 shows the pseudo-code for the tag program generation. First, the decoder function is generated. The initialization part involves generating code for initialization of the variables in the symbol table (e.g. operand declarations) and code for retrieving the opcode from the received RFID command. Then a switch statement is constructed. For each macro in the macros list, cases are added. For the response and all of its nested operands, C `structs` are created with the members as the operands and their widths. Then the symbol table is looked up and code is generated for setting the width of each operand to the value originally specified in the input macros specification. Next, code is generated for invoking the cyclic redundancy check (CRC) function to check the incoming command. The code for extracting the operands from the command into strings is generated. These strings can be manipulated within the behaviors corresponding to different primitives. This uses

```
// generate header file
define constants for opcode values
define functions for decoder, unpacking of integer command to strings,
           packing strings, crc checking, etc

// generate C file
// generate decoder function
for (each symbol in symbol table)
    generate code to initialize variables
generate code for getting opcode from command
make switch statement
for (each macro)
{
    create a case
    // add following statements to it-
        for (each operand in response)
            if (nested)
                generate structs with fields and width of fields
        generate a struct for response
        add each response operand and its width to it
        declare objects of each above struct
        for (each operand in response) {
            look-up width from symbol table, initialize width members in structs
            initialize operand members in above structs to zero
        }
        generate code for invoking crc function on command
        // unpacking
        for (each operand in response) {
            generate code for extracting each operand from command into a string
        }
        // behavior
        attach code for behavioral C
        // packing
        for (each operand in response) {
            generate code for packing each operand into a response string
        }
        generate code for calculating crc of response and append it to response
        invoke function to send response
}
create default case
generate other functions
```

Figure 22: Pseudo-code for Tag Program Generation.

the width values and position of each operand in the command. The user supplied behavior is attached to the corresponding macro case. Once the response code has been appended, the code for packing the operands into a response string is generated. This uses the width values and position of each operand in the response. Next, code is generated for invoking the CRC function on the response, attaching the value to the response and for invoking the function to send the response.

A commented code skeleton that follows the pseudo-code of the generated tag program is shown in Figure 23.

4.5 THE PROTOTYPE MICROPROCESSOR SYSTEM

The prototype microprocessor system was simulated using an Altera APEX 20 FPGA for *Smart Buffer* implementation and three different processor cores: the Intel StrongARM at 206 MHz [26], the Intel XScale 80200 at 733 MHz [27], and the 16-bit EISC microprocessor at 50 MHz from ADChips [28]. A prototype system was built with an Avnet development board, an EISC development board, and an *Air Interface* prototype board fabricated using PCB Express.

Figure 24 shows the interface between the EISC processor and the *Smart Buffer*. Thirteen of the EISC processor's I/O pins are used for communicating the command and response. When an complete RFID command is received, the *Smart Buffer* interrupts the processor. EISC reads the command from the FIFO queue in the *Smart Buffer*, the length of which is indicated by the *Smart Buffer*. This is implemented as an interrupt service routine in the EISC and is shown in Figure 25. Once the complete command is received, this routine invokes the decoder function, described in Section 4.4. When the decoder completes its execution, the tag's response is packed. The response is decomposed into packets and is sent to the *Smart Buffer* as shown in Figure 26. The processor then directs the *Smart Buffer* to transmit the RFID response to the analog front-end. These two routines are specific for the EISC microprocessor. When a different microprocessor is to be used for the RFID tag, these routines need to be updated by the application programmer.

```
void decoder() {
     // Declare all the operands
     long opcode;
     long prefix = -1;
     long type = -1;
     ..
     // Get the opcode from command
     StringCopy(opcode_str, command_array_ptr, 8);
     opcode = strtoul(opcode_str, &array_endptr, 2);
     ..
     switch(opcode) {
          case ICOL: {
                    typedef struct TAGSTATUS_STRUCT {
                         long modefield;
                         int size_modefield;
                         ...// next field
                    } TAGSTATUS_STRUCT;

                    typedef struct RESPONSE_STRUCT {
                         TAGSTATUS_STRUCT tagstatus;
                         int size_tagstatus;
                         long mesglen;
                         int size_mesglen;
                         ...// next field
                    } RESPONSE_STRUCT;
                    ..
                    TAGSTATUS.modefield = 0;
                    TAGSTATUS.size_modefield = 4;
                    ...// next field
                    RESPONSE.size_tagstatus = 16;
                    RESPONSE.mesglen = 0;
                    RESPONSE.size_mesglen = 8;
                    ...// next field
                    // Unpacking command:
                    // Extracting field prefix:
                    StringCopy(prefix_str, command_array_ptr, 8);
                    prefix_int = strtoul(prefix_str, &prefix_ptr, 2);
                    for(loop=0; loop<8; loop++)
                         *command_array_ptr++;
                    ...// extract next field
                    // Behavior of ICOL
                    TAGSTATUS.modefield = 0;
                    TAGSTATUS.reserved1 = 7;
                    ...// behavior of next field
                    // Packing response:
                    FieldToString(tagstatus_ptr, TAGSTATUS.modefield, TAGSTATUS.size_modefield);
                    array_ptr = PackFieldInArray(array_ptr, tagstatus_ptr);
                    ...// pack next field
                    SendResponse(array_ptr, word_size, num_words);
             }
          case IUDW: {
                    ..
             }
          ..// case next macro
          default: break;
      }
      // reset array for next command
      command_array_str[0] = '\0';
}
```

Figure 23: Outline of Compiler-generated Tag Program

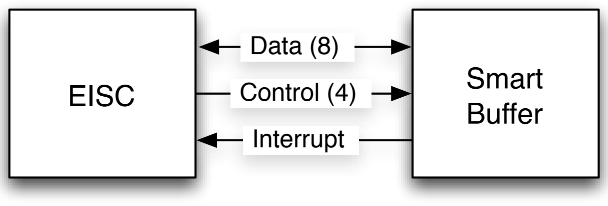


Figure 24: Interface Between the EISC processor and the *Smart Buffer*

The system was tested with a variety of automatically generated controller programs including anywhere from 1 to 14 RFID Primitives. While fitting additional primitives in the prototype system is theoretically possible, limitations of available memory on the board prevented a larger program size from being used.

The RFID compiler was used to generate three different programs: Program A with 24 RFID primitives, Program B with 12 primitives, and Program C with 4 RFID primitives. Experiments were conducted by executing 14 primitives of Program A, 1 primitive of Program A, 1 primitive of Program B, and 1 primitive of Program C.

The sim-panalyzer [29] and XTREM [30] tools were used to estimate the power dissipation of the compiler-generated microprocessor-based tag for the ARM-based cores. Sim-panalyzer is a cycle accurate, architecture-level power simulator built on the SimpleScalar processor simulator. XTREM is a SimpleScalar-based power and performance simulator tailored specifically for the Intel XScale micro-architecture. SimpleScalar's sim-profile tool was used to obtain ARM instruction and instruction class profiles for the RFID tag programs.

Because an instruction set simulator was not available for the EISC processor to measure the execution time, the application was run on the development board and a pin output was set from low to high upon each iteration. The total duration was measured using an oscilloscope. The energy consumed by EISC is calculated as the average power consumption multiplied by the measured execution time. The EISC power estimate was provided by AD Chips [28], which is estimated to be within about 10% accuracy of an instruction level power estimation approach [31].

```
void EXT_IRQ1_ISR()
{
     _PIO0_LDAT = 0x0001;                    // Read signal
     _PIO1_MOD  = 0x0000;                    // Write PIO1 [15:0]
     _PIO1_LDAT = 0x0000;                    // Initial
     _PIO1_MOD  = 0x0002;                    // Write PIO1 [15:2] and [0];
                                             // PIO1 [1]: Open collector output
     _PIO1_LDAT = 0x0040;                    // PULL
     _PIO1_MOD  = 0x0002;                    // PIO1 [1]: Open collector output
     empty      = (_PIO1_EDAT & 0x0002) >> 1;  // Reading only the empty bit.
                                                // Discarding everything else.
     command_array_ptr = command_array_str;
     while (((_PIO1_EDAT & 0x0002) >> 1) == 0)
     {
          _PIO1_MOD  = 0x0002;               // Write PIO1 [15:2] and [0]
          _PIO1_LDAT = 0x0044;               // PULL
          _PIO1_MOD  = 0xFF02;
          r_data_1   = _PIO1_EDAT >> 8;

          byte_ptr = byte_str;               // Convert byte to bit-string
          FieldToString(byte_ptr, r_data_1, 8);

          byte_ptr = byte_str;               // Pack bit_string into command array
          command_array_ptr = PackFieldInArray(command_array_ptr, byte_ptr);

          _PIO1_MOD  = 0x0002;               // Write PIO1 [15:2] and [0]
          _PIO1_LDAT = 0x0040;               // PULL
     }
     // Decode and Execute Command
     decoder();

     _PIO1_MOD  = 0x0000;                    // Write PIO1 [15:0]
     _PIO1_LDAT = 0x0000;                    // Initial
     _ISR_END  = EXT1_INT_Clr;
}
```

Figure 25: Interrupt Service Routine to Read the Incoming Command from the *Smart Buffer*

```
void SendResponse(char *array_ptr, int word_size, int num_words)
{
     int outputword, i, loop;
     char buffer_str1[50], buffer_str0[50];
     char *array_endptr;
     char Tx00_str[9] = {"00000000"};
     char Tx04_str[9] = {"00000100"};
     char *Tx00_ptr;
     char *Tx04_ptr;

     Tx00_ptr  = Tx00_str;
     Tx04_ptr  = Tx04_str;
     _PIO1_MOD  = 0x0000;
     _PIO1_LDAT = 0x0000;    // Init

     for(i=0; i< num_words; i++)
     {
          StringCopy(buffer_str1, array_ptr, word_size);
          StringCopy(buffer_str0, array_ptr, word_size);
          outputword = strtoul(buffer_str1, &array_endptr, 2);

          strncat(buffer_str1, Tx00_ptr, word_size);
          for(loop=0; loop< num_words; loop++)
               *array_ptr++;
          outputword = strtoul(buffer_str1, &array_endptr, 2);

          _PIO1_LDAT = outputword;

          strncat(buffer_str0, Tx04_ptr, word_size);
          outputword = strtoul(buffer_str0, &array_endptr, 2);

          _PIO1_LDAT = outputword;
          _PIO1_LDAT = 0x0000;
     }

     _PIO1_MOD  = 0x0000;    // Direction
     _PIO1_LDAT = 0x0000;    // Init
     _PIO1_LDAT = 0x0066;    // Transmit
     _PIO1_LDAT = 0x0000;    // Init
}
```

Figure 26: Routine to Send the Response to the *Smart Buffer*

4.6 RESULTS

Figures 27 and 28 show the power and energy consumption results of the compiler-generated tag program on the Intel StrongArm SA -110 [26], Intel XScale 80200 [27], and the ADChips 16-bit EISC [28].

These results show the power consumption only during the active phase of the RFID transaction (e.g. the time after the entire packet is received by the smart buffer when the packet is processed and the response is generated by the smart buffer). During this time, the *smart buffer* is active and its ASIC implementation consumes 0.29mW, as shown in [25]. This power is negligible compared to the processor power.

The frequencies of operation used are 206.40MHz for StrongArm, 733MHz for XScale, and 50 MHz for EISC. Both the ARM-based processors operate in the 250-400 mW range, while the XScale uses significantly less energy (10s versus 100s of μJ). The EISC processor uses an order of magnitude less power than the ARM-based cores, but operates at a much slower clock speed. However, the energy consumed is still less than half of XScale.

It can be seen that the power consumption of XScale is less than that of StrongArm though they both implement the ARM Instruction Set Architecture. This is because the XScale family of microprocessors uses deep pipelines and micro-architectural optimizations for high performance [27]. Further, the reduced power consumption and greater clock speed of XScale 80200 result in its far lower energy consumption.

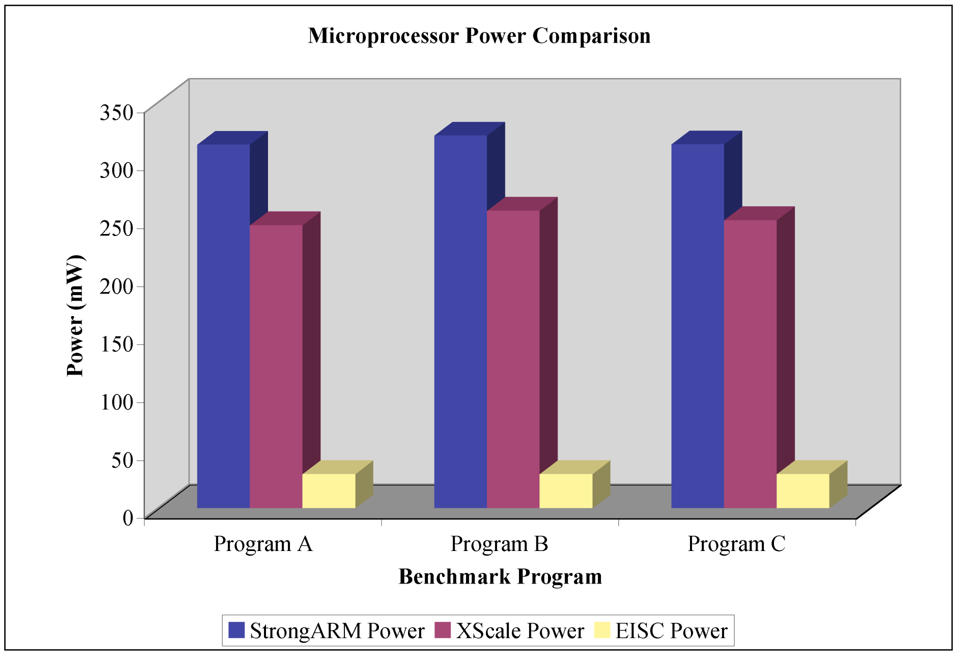


Figure 27: Power consumption results for microprocessor-based RFID tags.

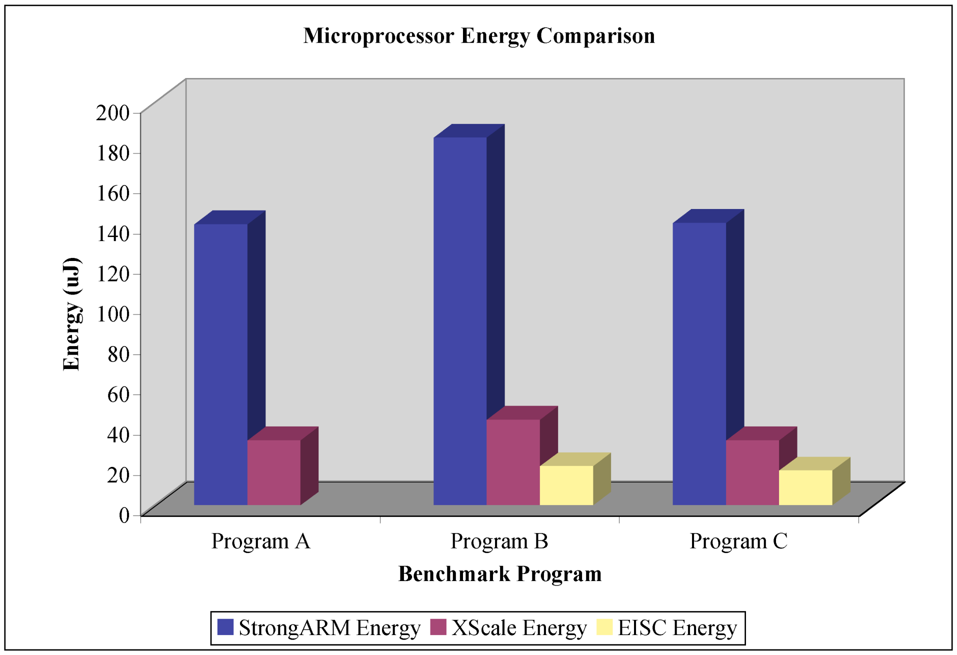


Figure 28: Energy results for microprocessor-based RFID tags during response generation.

5.0 HARDWARE RFID COMPILER WITH VHDL BEHAVIOR

The overhead of using a microprocessor-based controller for the RFID tag is considerable. For example, Intel StrongARM and XScale processors operate in the hundreds of mW range. While the *smart buffer* is intended to alleviate much of the controller's power consumption by putting the processor to sleep, the ARM-based processors can require hundreds of instructions to be executed to generate the response for a single primitive. This can result in significant energy usage even with the smart buffer. Ideally, the EISC processor would be used for low-power purposes, however, system memory is a limitation.

A hardware-based solution may improve both the power and capacity of the controller. This chapter presents extensions to the RFID compiler design flow to generate VHDL for targeting a hardware-based controller. The compilation flow the for the Hardware-based RFID tag is shown in Figure 29.

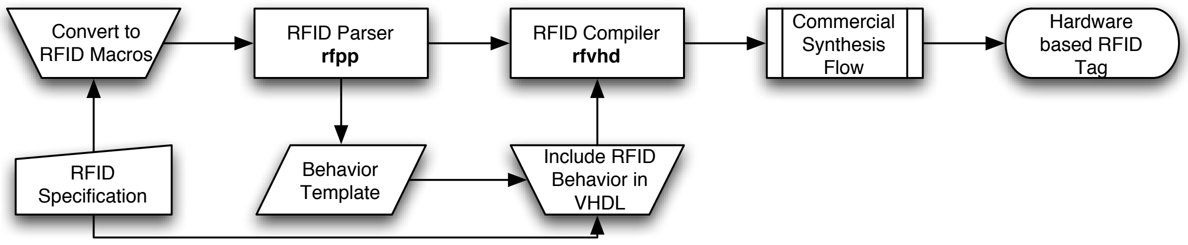


Figure 29: Hardware RFID Compiler with VHDL Behavior

5.1 VHDL BEHAVIOR-BASED FRONT-END

Since the code generated by the compiler needs to be synthesizable VHDL, the behavior for each primitive also needs to be input in VHDL. The RFID parser was modified so that it generates VHDL assignment statements in the templates after processing the original macros file described in Section 4.2.

The template generated for the *Collection* command (in Figure 14) is shown in Figure 30. The completed behavior for the same command is shown in Figure 31. The behavioral VHDL statements added by the user need to be synthesizable. The code segment for packet packing, unpacking, and decoding is automatically generated and output in VHDL rather than in C. Finally, the RFID compiler generates the tag controller VHDL design which is synthesized, mapped, placed, and routed for the target FPGA device using commercially available tools.

5.2 HARDWARE CONTROLLER-BASED BACK-END

PACT HDL (Power Aware Architecture and Compilation Techniques) is a compiler targeting the C language that produces synthesizable HDL usable for either FPGA designs or Application Specific ICs (ASICs) with a framework for both power and performance optimizations [32]. The VHDL abstract syntax tree (AST) of PACT HDL has been used in the RFID compiler back-end to create the RFID tag controller VHDL. The VHDL design so generated is industry-standard, and may be synthesized and profiled for power using commercial tools.

An outline of the automatic VHDL generation using the VHDL AST is shown in Figure 32. The highest VHDL data structure is a design file. Therefore, the back-end creates a DesignFileClass. A design file contains a design unit. A design unit contains an entity declaration and an architecture declaration. Classes are instantiated for each of these data types and added to the design file.

```
modefield =
reserved1 =
acknowledge =
reserved2 =
tagtype =
reserved3 =
userid =
battery =
mesglen =
interid =
tagid =
ownerid =
crc =
```

Figure 30: Template Generated for Behavior of *Collection* Command

```
modefield := "0000";
reserved1 := "111";
if (commandValid = '1') then
    acknowledge := '0';
else
    acknowledge := '1';
end if;
reserved2 := "11";
tagtype := "010";
reserved3 := '1';
userid := '0';
battery := '0';
mesglen := X"0E";
crc := X"0000";
```

Figure 31: Behavior of *Collection* Command After User Input

A VHDL entity declaration contains the description of the communication between that entity and the external blocks. The top-level entity contains the ports (input and output declarations), a global symbol table of VHDL signals, and multiple process nodes corresponding to the tag state machine. The process nodes contain local symbol tables with VHDL variables and statement nodes. Statement nodes such as assignments, if statements, etc., are made of expressions and operators.

A new UserDefinedStatementClass, descended from SequentialStatementClass, was added to the VHDL AST to support the synthesizable VHDL statements input by the user to describe the behavior of the primitives.

5.3 IMPLEMENTATION OF PRIMITIVES

The Hardware RFID Compiler was validated by implementing the primitive logic for different hardware targets. The system was tested with 2 primitives up to 40 primitives. 40 primitives was selected as the maximum because it provided room for all the primitives of the ANSI-256 and the ISO 18000-7 standards, with room for several customized primitives. The commands from ISO 18000-7 and ANSI-256 have been summarized in Figure 33 and Figure 34 respectively. The commands are explained in Section 4.1. The corresponding hardware implementations are discussed in Sections 5.3.1 and 5.3.2.

5.3.1 FPGA-Based Implementation

The prototype FPGA-based system was implemented in simulation using a Xilinx Spartan 3 XC3S400 FPGA to implement the smart buffer and a Xilinx Coolrunner II XC2C512 CPLD to implement the primitive logic programmed into the microprocessor in the previous system. The primitive logic was also implemented in simulation using an Actel Fusion AFS090 FPGA. The entire system was also loaded onto a single Spartan 3 XC3S400 and tested in hardware. The system was tested with 2 primitives up to 40 primitives.

```
// Create design file
DesignFileClass *df = new DesignFileClass();
// Create top level design unit
DesignUnitClass *du = new DesignUnitClass(new LibraryClauseClass("IEEE"));
// Add use clauses
UseClauseClass *uc = new UseClauseClass(selName);
du->addClause(useClause); //...
// Create entity
EntityDeclrClass *entity = new EntityDeclrClass(entityId, decs);
// Add ports to entity
InterfaceClass *ports = new InterfaceClass(SIG, portId, portMode, portSubtype);
entity->addPortInterface(ports); //...
// Add entity  to top level design
du->setLibUnit(entity);
// Create architecture
ArchitectureDeclrClass *architecture = new ArchitectureDeclrClass(behavior, id);
// Add signals, etc
SignalDeclrClass *sigDec = new SignalDeclrClass(idlist, subType);
architecture->addBlock(sigDec);
// Create process
ProcessStatementClass *processStmt = new ProcessStatementClass(taglogic);
// Add sensitivity
processStmt->addSensitivity(id); //...
// Add declarations
while(symbolTable) {
      IdentifierClass *procDecId = new IdentifierClass(symbolTable->operandName);
      range = new DiscreteRangeClass(DOWNTO, d0, new DecimalClass(decList->value));
      subtype = new SubtypeIndicationClass(STD_LOGIC_VECTOR, range);
      VariableDeclrClass *varDec = new VariableDeclrClass(procDecId, subtype);
      processStmt->addDeclaration(varDec); //...
}
// Create case statement
CaseStatementClass *caseStmt = new CaseStatementClass(currentstate);
// Add case body
CaseBodyClass *caseBody = new CaseBodyClass(s0);
// Add statements to case body
sigAssignStmt = new SignalAssignmentStatementClass(responseReady, prWaveform);
caseBody->addSequential(sigAssignStmt);
// Parse and Add user supplied behavior statements to case body //...
// Add packing logic statements // ...
// Add completed case to process
processStmt->addSequential(caseStmt);
// Add completed process to architecture
architecture->addConcurrent(processStmt);
// Add completed architecture to design unit
du->addClause(architecture);
// Add design unit to design file
df->addDesignUnit(du);
```

Figure 32: Outline of Automatic VHDL Design Generation Using VHDL AST.

```
Collection
Collection with data
Collection with user id
Sleep
Read tag status
Firmware version
User ID length read
User ID length write
User ID read
User ID write
Owner ID read
Owner ID write
Model number
Set password
Set password protect
Unlock
Write memory
Read memory
```

Figure 33: Commands from ISO 18000 Part 7.

```
Collection
Sleep
Sleep all but
Sleep all but group
Get tag status
Get version
Set group code
Get group code
Get error
Clear error
Beeper on
Beeper off
Write memory
Read memory
```

Figure 34: Commands from ANSI NCITS 256-2001.

Table 1: Area and performance result for implementing the primitive logic on a Coolrunner II XC2C512

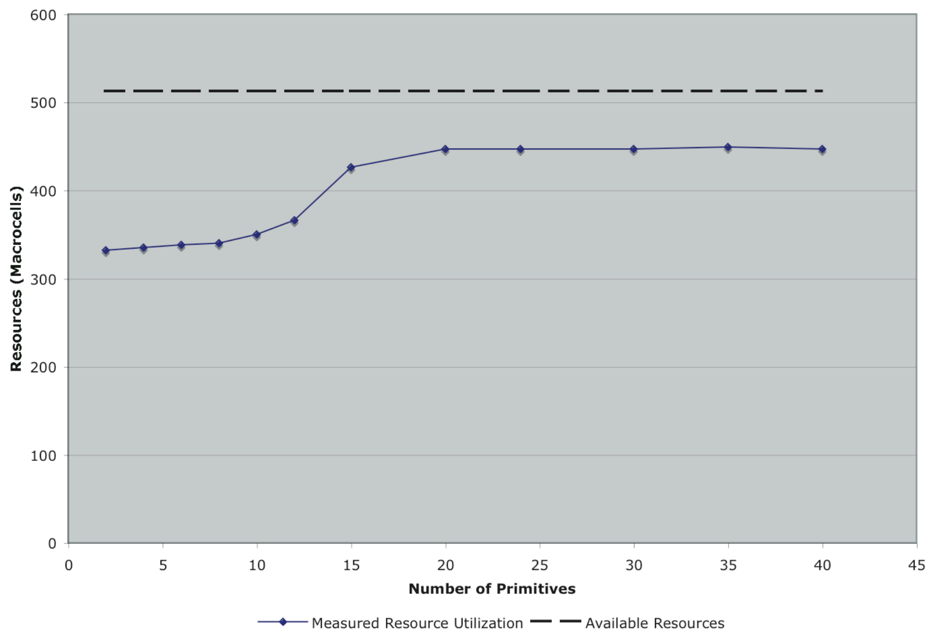
Primitives	**2**	**4**	**6**	**8**	**10**	**12**	**15**	**20**	**24**	**30**	**35**	**40**
Macrocells	332	335	338	340	350	366	426	447	447	447	449	447
% used	66%	66%	67%	67%	69%	72%	84%	88%	88%	88%	88%	88%
Product Terms	444	477	514	506	477	552	772	953	993	1106	1181	1213
% used	25%	27%	29%	29%	27%	31%	44%	54%	56%	62%	66%	68%
Registers	262	267	271	271	283	307	422	443	443	443	443	443
% used	52%	53%	53%	53%	56%	60%	83%	87%	87%	87%	87%	87%
Func. Block Inputs	360	379	408	391	338	396	611	767	801	870	900	914
% used	29%	30%	32%	31%	27%	31%	48%	60%	63%	68%	71%	72%
FMax (MHz)	49	41	41	40	49	41	41	29	33	33	26	30

The area and performance results for the user defined primitive logic are shown in Table 1 for the Coolrunner II XC2C512 CPLD and Table 2 for the Actel Fusion AFS090 FPGA. They are summarized against the available resources in Figure 35. Figure 36 shows the resource utilization of the user defined primitive logic and the smart buffer logic on the Spartan 3 prototype system.

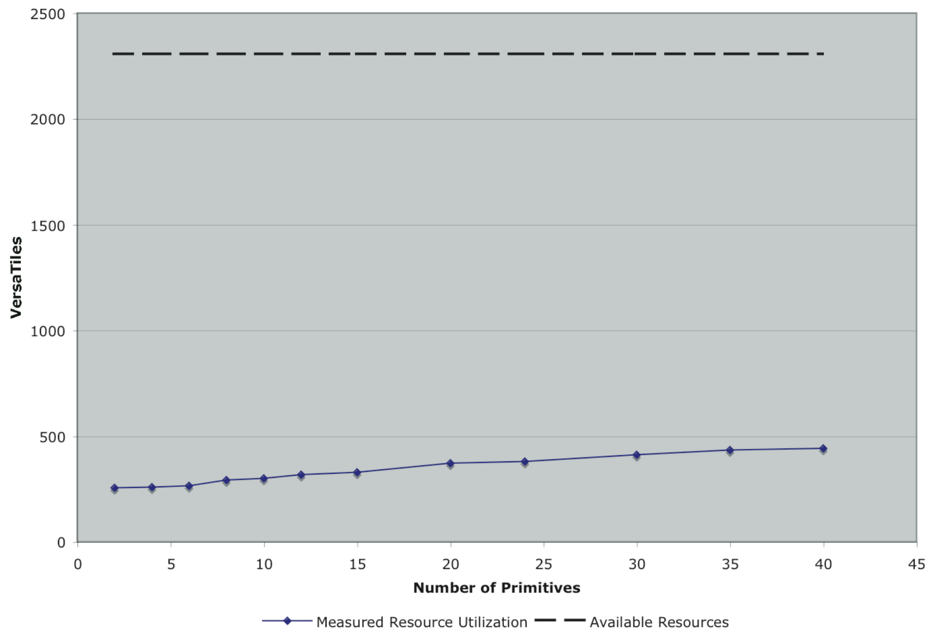
From Figures 35 and 36, it can be seen that the logic required by the RFID tag response generation fits into all the three devices. The Coolrunner II is a small, low-power device and has 512 macrocells. They are utilized by up to 88% by the 40 primitive tag program. The Actel Fusion device has 2304 VersaTiles which are utilized by up to 19%. The Spartan 3 provides plenty of capacity, with 7168 Look Up Tables (LUTs). The tag program along with the smart buffer logic utilizes only about 37% of this.

Table 2: Area for implementing the primitive logic on an Actel Fusion AFS090.

Primitives	**2**	**4**	**6**	**8**	**10**	**12**	**15**	**20**	**24**	**30**	**35**	**40**
Cells	315	317	324	351	359	376	388	430	439	470	493	501
VersaTiles	256	258	265	292	300	317	329	371	380	411	434	442
% used	11.1%	11.2%	11.5%	12.7%	13.0%	13.8%	14.3%	16.1%	16.5%	17.8%	18.8%	19.2%



(a) Primitive logic on a Coolrunner II.



(b) Primitive logic on an Actel Fusion.

Figure 35: Resource utilization of user defined logic for FPGA-based prototype systems.

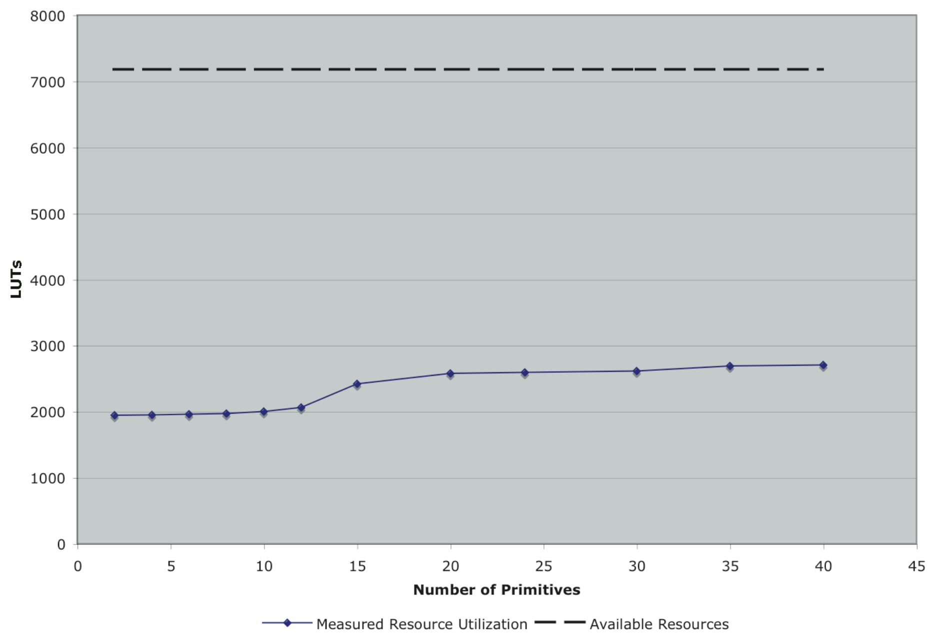


Figure 36: Resource utilization of user defined primitive logic and smart buffer on a Spartan 3 prototype system.

The power consumption of the Xilinx FPGA devices including the Spartan 3 and Coolrunner II was estimated using Xpower from Xilinx. For the Actel Fusion FPGA, the tool used was SmartPower from Actel. Switching statistics for the tools were generated from cycle-accurate, post place and route simulations of actual test data.

While the Spartan 3 certainly provides plenty of capacity, its power consumption, albeit potentially lower than the ARM-based processors, is still not as low as desired. Unfortunately, much of this is due to the quiescent power of 92 mW. In order to further reduce power, the Fusion and Coolrunner II implementations were explored. For these systems, the smart buffer is not implemented in reconfigurable logic. Only the user defined primitive controller generated by the compiler is implemented in the CoolRunner / Fusion devices. The power and energy consumption of the Coolrunner2-based RFID tag controller are displayed in Table 3. The power consumption results for the Fusion-based RFID tag are summarized in Figure 37.

The Actel Fusion FPGA consumes 12.98 mW for the 40 primitive tag program. Its quiescent power is 7.5 mW. However, for the same primitives running on a Coolrunner II FPGA, the required power drops to 1.1 mW. Additionally, the Coolrunner quiescent power is approximately 50 μW, which is a reasonable power consumption for idle modes of an active RFID tag.

5.3.2 ASIC Implementation

The RFID system designer can choose the between FPGAs and ASICs as the implementation medium for designing tags, in the early stages. We described the significant differences in design times and flexibilities of these mediums in Section 3.2. Metrics such as area (which affects the cost), performance and power consumption differences also need to be considered to assess whether an FPGA implementation is feasible. Of these, the primary factors for the design of an RFID tag are low power and low area.

Recently, there have been a number of attempts to quantify the difference between FPAGs and standard cell ASICs in terms of area, performance and power consumption. A recent work compares a 90 nm CMOS SRAM-programmable FPGA and a 90 nm CMOS standard

Table 3: Power and energy results for implementing the primitive logic on a Coolrunner II XC2C512

Number of	**Power (mW)**			**Energy (μJ)**
Primitives	**Dynamic**	**Quiescent**	**Total**	**Total**
2	1.06	0.05	1.11	0.00111
4	1.06	0.05	1.11	0.00111
6	1.06	0.05	1.11	0.00111
8	1.07	0.05	1.12	0.00112
10	1.06	0.05	1.11	0.00111
12	1.06	0.05	1.11	0.00111
15	1.24	0.05	1.29	0.00129
20	1.24	0.05	1.29	0.00129
24	1.24	0.05	1.29	0.00129
30	1.24	0.05	1.29	0.00129
35	1.24	0.05	1.29	0.00129
40	1.24	0.05	1.29	0.00129

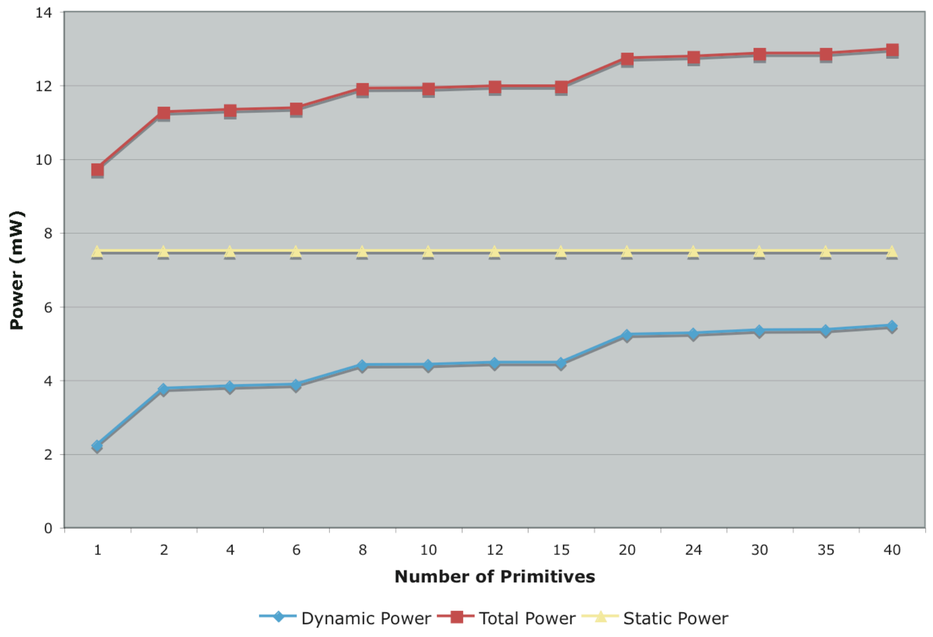


Figure 37: Power consumption for implementing the primitive logic on Actel Fusion.

Table 4: Area for implementing the primitive logic on a 0.16μm ASIC. ASIC area is in $100\mu m^2$.

Primitives	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**15**	**20**	**24**	**30**	**35**	**40**
Cells	3761	3809	3836	3841	3835	3859	3961	3990	4140	4170	4235	4264	4298
Area	1.076	1.092	1.097	1.098	1.097	1.105	1.116	1.121	1.158	1.161	1.174	1.182	1.187

cell technology [33]. According to this work, the dynamic power consumption of FPGAs is about 12 times more than ASICs. Further, the area required to implement the same logic on FPGAs is about 40X more than on ASICs when only combinational logic and flip-flops are used. When blocks such as multiplier / accumulators and memories are used, the area gap is reduced to 20 X. The performance is about 4X worse in FPGAs.

Thus, implementing the primitives in ASICs may potentially save a significant amount of power and area of the RFID tag. In this section, we implement RFID primitives in ASICs to further explore this. These results can also be used for comparing the automatically generated designs to the commercial ASIC RFID tags. Up to 40 RFID primitives generated using the RFID compiler were implemented in custom cell-based ASIC hardware at 0.16 μm. The area results are displayed in Table 4. The power consumed by the various hardware implementations is summarized in Figure 38.

The energy consumed by the ASIC implementation per transaction is 0.07 nJ per transaction when active, which is comparable to the energy used by commercial active tags (for example, see [14]) which are implemented as ASICs. This is much less than the 13 nJ and 1.3 nJ consumed per transaction for tags using Fusion FPGA and Coolrunner II CPLD, respectively. Futher, the static power consumed by the ASIC implementation is 0.4 μW, which is about two orders of magnitude less than that of the low-power Coolrunner II CPLD. The area of the ASIC controller is less than 4300 cells for a 40 primitive controller, which is also comparable to the commercial active tags.

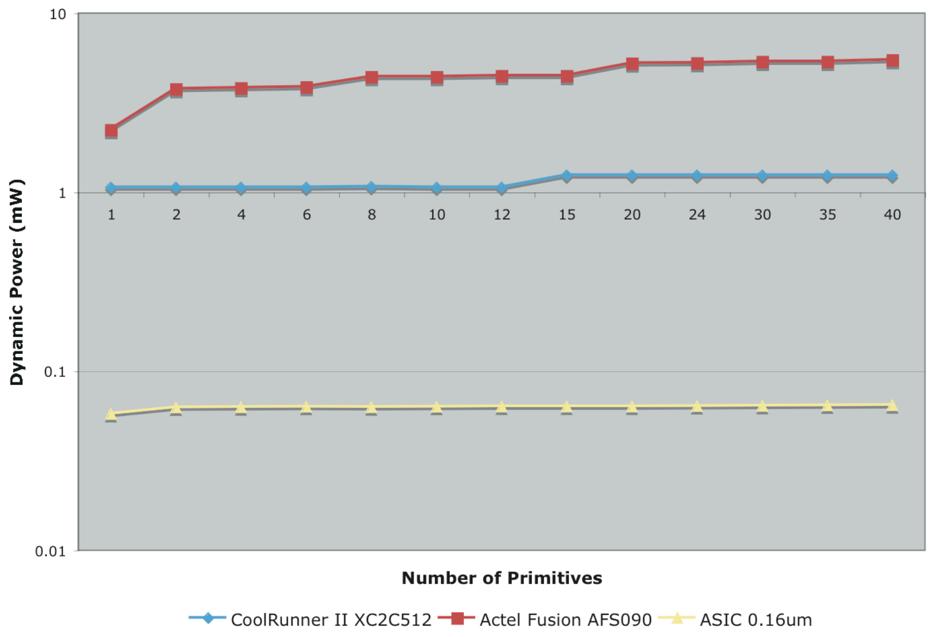


Figure 38: Power consumption comparison for implementing the primitive logic on a Coolrunner II XC2C512, an Actel Fusion AFS090, and 0.16μm ASIC.

6.0 HARDWARE RFID COMPILER WITH C BEHAVIOR

The software RFID compiler (described in Chapter 4) generates a complete application in C that is compiled for the target microprocessor in the RFID Tag. While the behavior is specified in C by the user, much of the remaining C code is automatically generated from the RFID macros. For the initial hardware RFID compiler (described in Chapter 5), this automatically generated code segment (e.g. for packet handling, etc.) was output in VHDL rather than C. In many ways, generating the VHDL code from the RFID macros is actually more natural as VHDL handles arbitrary bit widths more naturally than C/C++.

Because C is a significantly more universally known language than VHDL or Verilog, it is desirable to continue having the end-user specify the primitive behaviors for the RFID Tag in C code. This requires that the C code be converted in synthesizable hardware code. Preferably, this code would also be as simple as possible and optimized for power.

In order to synthesize primitives into VHDL, a new version of the hardware RFID compiler was developed based on both the automated VHDL generation described in Chapter 5 and the SuperCISC compilation flow described in [34, 35]. The hardware RFID compiler can read primitive behavioral descriptions in non-modified ANSI-C and generate entirely synthesizable VHDL for combinational implementation. These combinational blocks are combined with the automatically generated packet packing, unpacking, and decoding VHDL and synthesized for the reprogrammable hardware target. The RFID compiler automatically handles the generation of a finite state machine (FSM) controller for the final tag hardware.

6.1 C BEHAVIOR-BASED FRONT-END

The compilation flow the for the hardware-based RFID tag is shown in Figure 39. The RFID compiler processes the original RFID macros (previously shown in Figure 16) to generate C template files to specify the behavior. The behavior for each primitive is specified using ANSI-C. The back-end converts this into synthesizable VHDL and generates the final tag VHDL.

6.2 BACK-END INTEGRATED WITH SUPERCISC COMPILER

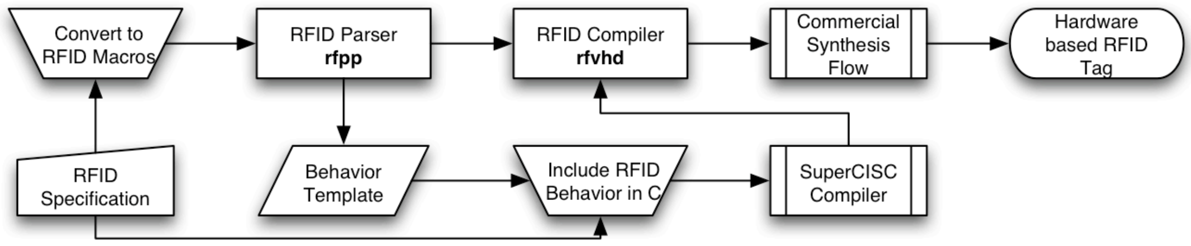


Figure 39: Hardware RFID Compiler with C Behavior

The completed behavior in ANSI-C of each primitive is marked for hardware creation as shown in Figure 40. Each such behavior is fed into the SuperCISC compiler [34, 35]. The SuperCISC compiler converts each primitive behavior into a combinational hardware block.

Figure 41 shows the conversion of the input C code into combinational hardware. First the C code is represented in a control and data flow graph (CDFG) representation as shown in Figure 41(a) for the *Collection* (`icol`) primitive. CDFGs are commonly used within compilers for transformations and optimizations. Many behavioral synthesis tools also use CDFGs as their internal representation [36, 37]. The CDFG shown in Figure 41(a) has the control flow graph (CFG) on the far left. The edges between each block represent control dependencies. Generally, control dependencies indicate that a decision must be made to decide which hardware becomes active next. Often, cycle boundaries are created due to the

```
int main()
{
  int commandValid = 1;
  // fields in the command
  int prefix, type, ownerid, interid, comcode, siz, res, crc;
  // fields in the response
  int modefield, reserved1, acknowledge, reserved2, tagtype,
      reserved3, userid, battery, mesglen, tagid;
  #pragma HWstart;
  modefield = 0;
  reserved1 = 7;
  if (commandValid)
    acknowledge = 0;
  else
    acknowledge = 1;
  reserved2 = 3;
  tagtype = 2;
  reserved3 = 1;
  userid = 0;
  battery = 0;
  mesglen = 14;
  #pragma HWend;
  return (modefield + reserved1 + acknowledge + reserved2 +
          tagtype + reserved3 + userid + battery + mesglen);
}
```

Figure 40: Behavior of *Collection* Command Marked for Hardware Creation.

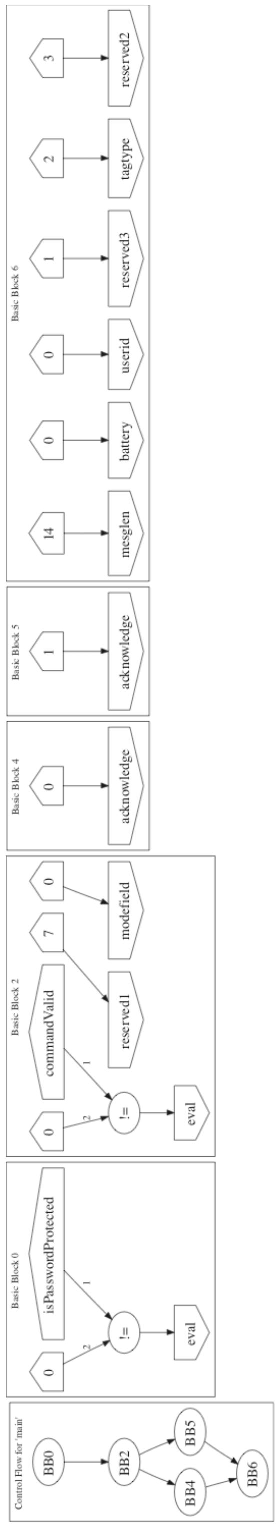
control dependencies during synthesis of CDFGs. Each block in the CFG is a basic block containing a data flow graph (DFG). All of the basic blocks in the CFG are shown to the right of the CFG. Edges in the DFG represent data flow dependencies and can be implemented with combinational logic only (e.g. no cycle boundaries) during behavioral synthesis.

The SuperCISC compiler translates the CDFG into an entirely combinational representation called a super data flow graph (SDFG). This process takes advantage of several well known compiler transformations such as loop unrolling and function inlining as well as a technique called *hardware predication* to convert all control dependencies into data dependencies creating an entirely combinational representation. The SDFG for the `ionw` is shown in Figure 41(b). Because the SuperCISC technique removes the need for many potentially high-power consumption sequential constructs such as registers and clock trees, SDFG-based hardware implementations are extremely power efficient [38].

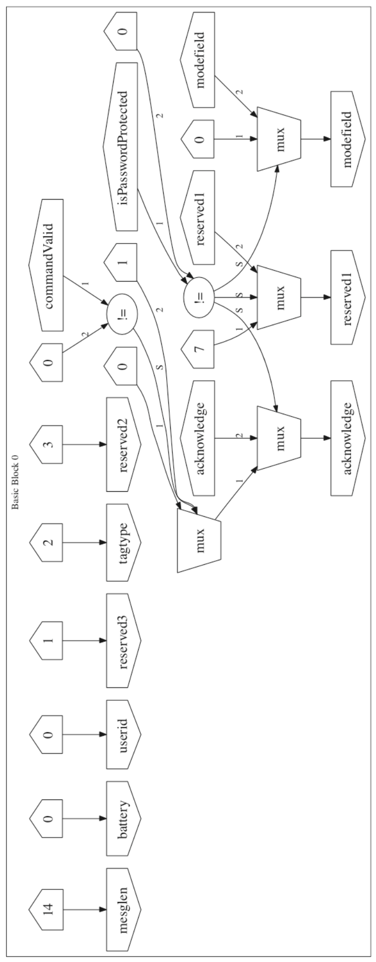
Next, the SuperCISC compiler translates the SDFG of each primitive into a VHDL description using the VHDL AST data structures described in Section 5.2. The DesignFileClass node has a design unit, with an entity node describing its ports and an architecture node describing its function. The generated VHDL for the *Collection* (`icol`) primitive is shown in Figure 42.

The SuperCISC compiler was modified by adding a function, *printComponentDetails()*, that walks through the DesignFileClass node, reads the ports in the entity, and prints the information to a text file. An outline of this function is shown in Figure 43.

The automatic generation of the packet packing, unpacking, and decoding VHDL uses the VHDL AST data structures and is similar to the automatic VHDL generation described in Section 5.2. However, in this case, the behavior components need to be attached to the RFID tag DesignFileClass node. For this, the RFID compiler parses the text file generated by the *printComponentDetails()* function and constructs the corresponding component declarations, instances and port mappings in the tag controller VHDL. An outline of the corresponding function, *addBehavComponent()*, is shown in Figure 44. The resulting synthesizable VHDL is synthesized, mapped, placed, and routed for the target FPGA or ASIC using commercially available tools.



(a) Control and data flow graph.



(b) Super data flow graph.

Figure 41: Synthesis process for the *Collection* (`icol`) command.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
use IEEE.std_logic_arith.all;
entity icol is
port (
    signal commandValid: IN std_logic_vector(31 DOWNTO 0);
    signal modefield_out: OUT std_logic_vector(31 DOWNTO 0);
    signal reserved1_out: OUT std_logic_vector(31 DOWNTO 0);
    signal acknowledge_out: OUT std_logic_vector(31 DOWNTO 0);
    signal reserved2_out: OUT std_logic_vector(31 DOWNTO 0);
    signal tagtype_out: OUT std_logic_vector(31 DOWNTO 0);
    signal reserved3_out: OUT std_logic_vector(31 DOWNTO 0);
    signal userid_out: OUT std_logic_vector(31 DOWNTO 0);
    signal battery_out: OUT std_logic_vector(31 DOWNTO 0);
    signal mesglen_out: OUT std_logic_vector(31 DOWNTO 0));
end entity icol;
architecture behavior of icol is
signal con2: std_logic_vector(31 DOWNTO 0):= "00000000000000000000000000000000";
signal con4: std_logic_vector(31 DOWNTO 0):= "00000000000000000000000000000111";
signal sig6: std_logic;
.......
component is_not_equal_to
port ( .. );
component mux
port ( .. );
begin
I0:component is_not_equal_to
port map (A => commandValid, .. );
I1:component mux
port map (A => con10, .. );
process (con2, con4, sig25, con14, con16, con18, con20, con22, con24)
begin
    modefield_out <= con2;
    reserved1_out <= con4;
    acknowledge_out <= sig25;
    reserved2_out <= con14;
    tagtype_out <= con16;
    reserved3_out <= con18;
    userid_out <= con20;
    battery_out <= con22;
    mesglen_out <= con24;
end process;
end architecture behavior;
```

Figure 42: VHDL Design for the *Collection* (`icol`) command.

```
void printComponentDetails(DesignFileClass *dfin)
{
  list<DesignUnitClass *> *dulist = dfin->getDesignList();
  if (dulist->size() > 0) {
     dulist->pop_back();
     // Get each design unit and its entity
    DesignUnitClass *du_entity = dulist->back();
    LibraryUnitClass *lb = du_entity->getLibUnit();
    EntityDeclrClass *ed = (EntityDeclrClass *)lb;
    // Open a file for each entity
     FILE *fp = fopen((ed->getName())->getStr(), "w");
    fprintf(fp, "%s\n",(ed->getName())->getStr());
    fprintf(fp, "portlist:\n");
    if ((ed->getPortList()) != 0) {
         InPtr = "IN  ";
         OutPtr = "OUT ";
         list<InterfaceClass *> *plist = ed->getPortList();
         list<InterfaceClass *>::iterator listIter;
         InterfaceClass *front = plist->front();
         // Traverse the list of ports
         for(listIter = plist->begin(); listIter != plist->end(); ++listIter) {
              list<IdentifierClass *> *idlist = (*listIter)->getId();
              IdentifierClass *portid = idlist->front();
              ModeClass *mode = (*listIter)->getMode();
              outPortName = (char *)malloc(strlen(portid->getStr()) + 1);
              portc = portid->getStr();
              // Write name of port, width and in/out info to file
              if ((mode->getKey()) == OUT) {
                   outPortName = strncpy(outPortName,portc,(strlen(portc)-strlen("_out")));
                   outPortName[strlen(portc) - strlen("_out")] = '\0';
                   portPtr = strcpy(portPtr, OutPtr);
                   portPtr = strcat(portPtr, outPortName);
                   if (*listIter == front)
                        fprintf(fp, "%s\n",portPtr);
                   else if (strcmp(outPortName, prev_portc))
                        fprintf(fp, "%s\n",portPtr);
                       prev_portc = outPortName;
                     }
              else if ((mode->getKey()) == IN) {
                   inPortName = portid->getStr();
                   portPtr = strcpy(portPtr, InPtr);
                   portPtr = strcat(portPtr, inPortName);
                   if (*listIter == front)
                        fprintf(fp, "%s\n",portPtr);
                   else if (strcmp(inPortName, prev_portc))
                        fprintf(fp, "%s\n",portPtr);
                   prev_portc = inPortName;
              }
          }
      }
   }
}
```

Figure 43: SuperCISC Function That Prints Details of Each Entity.

```
void addBehavComponent(FILE *behavFile, ArchitectureDeclrClass *architecture) {
  // Create a Component Declaration Class, Component Instantiation Class
  ComponentDeclrClass *component1;
  ComponentInstantiationStatementClass *compInstance;
  do {
    c = fgets(line_ptr, 200, behavFile);
     ..
     if (c != NULL) { ..
      if (j == 1) { ..
          compNamePtr = strcpy(compNamePtr, line_ptr);
          // Construct Component Declaration
          component1 = new ComponentDeclrClass(compName, compGlist, compPlist);
          // Construct Component Instantiation Class
          compInstance = new ComponentInstantiationStatementClass(compInName, instanceName);
       }
      else if (j == 2) {
          indicatePtr = strcpy(indicatePtr, line_ptr);
          ..
       }
      else {
         if (portsFollow) {
           if (!strncmp(portPtr, InPtr, 3)) {
               compMode =  new ModeClass(IN);
               for (k = 0; k < 4; k++)
                    *portPtr++;
               portNamePtr = strcpy(portNamePtr, portPtr);
               ..
               InterfaceClass *compInterface = new InterfaceClass(SIG, compId, compMode,
                                                                   compSubtype);
               // Add port to component declaration
               component1->addPort(compInterface);
               ..
               AssociationClass *compPortAssoc = new AssociationClass(compId, compId);
               // Add port to component instance
               compInstance->addPort(compPortAssoc);
            }
           else if (!strncmp(portPtr, OutPtr, 3)) {
              compMode =  new ModeClass(OUT);
              ..
             // Add port to component declaration
              // Add port to component instance
            }
         }
       }
    }
  } while (c != NULL);
  // Add component to architecture
  architecture->addBlock(component1);
  // Add instance to architecture
  architecture->addConcurrent(compInstance);
}
```

Figure 44: RFID Compiler Function That Adds Components to the RFID Tag VHDL AST.

6.3 IMPLEMENTATION OF PRIMITIVES

Up to 24 RFID primitives generated using the RFID compiler were implemented in custom cell-based ASIC hardware at 0.16 μm. These include the 16 primitives of the ISO 18000-7 and the 8 primitives of the ANSI standards, which were described previously in Section 4.1. The tool used for synthesis and area estimates is Design Compiler. Power was estimated using PrimePower. The area and power results are displayed in Table 5. As expected, the tag design uses more standard cells and increases in area as new primitives are added. The dynamic power increases slightly from the design with one primitive to the design with two primitives, but does not increase significantly as more primitives are added.

Table 5: Area and Dynamic Power for implementing the primitive logic on a 0.16μm ASIC. ASIC area is in 100μm^2. Dynamic Power is in mW. Quiescent Power <0.4μW.

Primitives	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**15**	**20**	**24**
Cells	3859	4140	4170	4207	4251	4306	4348	4389	4490	4525
Area	1.107	1.142	1.159	1.172	1.178	1.194	1.215	1.222	1.264	1.270
Power	0.075	0.082	0.083	0.085	0.085	0.085	0.085	0.085	0.085	0.086

6.3.1 Comparison With Hardware RFID Compiler With VHDL Behavior

The ASIC designs implemented using the two versions of the hardware RFID compiler are compared in terms of their areas and power consumption. The results are shown in Figures 45 and 46. There is an increase in area and power consumed when C behavior is used instead of direct VHDL. For the tag with a single primitive, there is a nominal increase in area of 2.91%. As the primitives are added, the area increase percentage rises slightly, and is 9.37% for the tag with twenty four primitives. The dynamic power increase percentage ranges from 29.38% for the tag with a single primitive to 34.48% for the tag with twenty four primitives.

The increase in area and power is in part due to how design compiler does resource sharing. It is also partly due to some extra logic introduced by the hardware predication pass of the SuperCISC Compiler. Design compiler does allow resource sharing through use of specialized controls, which provide an opportunity to reduce this overhead.

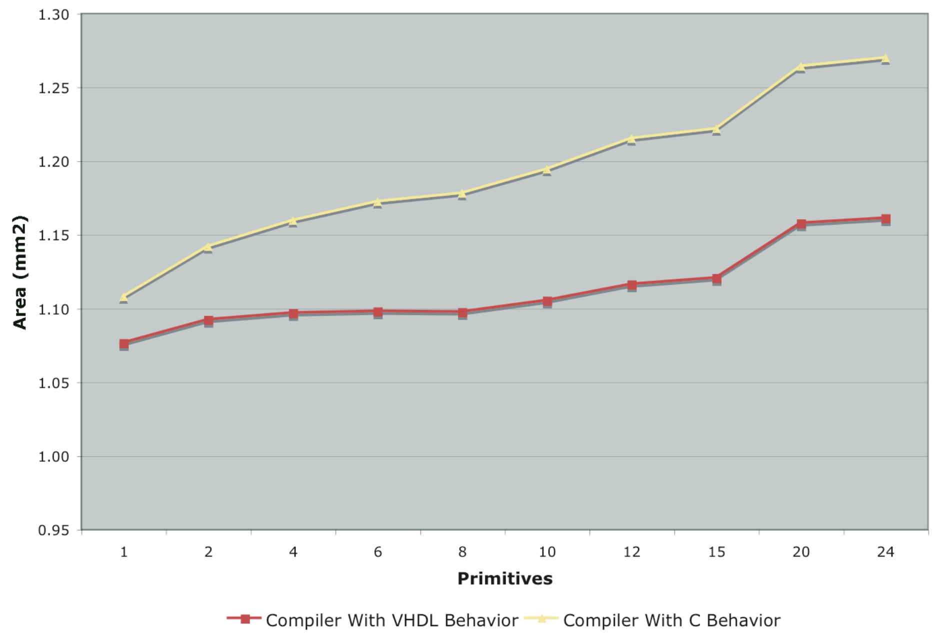


Figure 45: Area Comparison of the Two Hardware RFID Compilers.

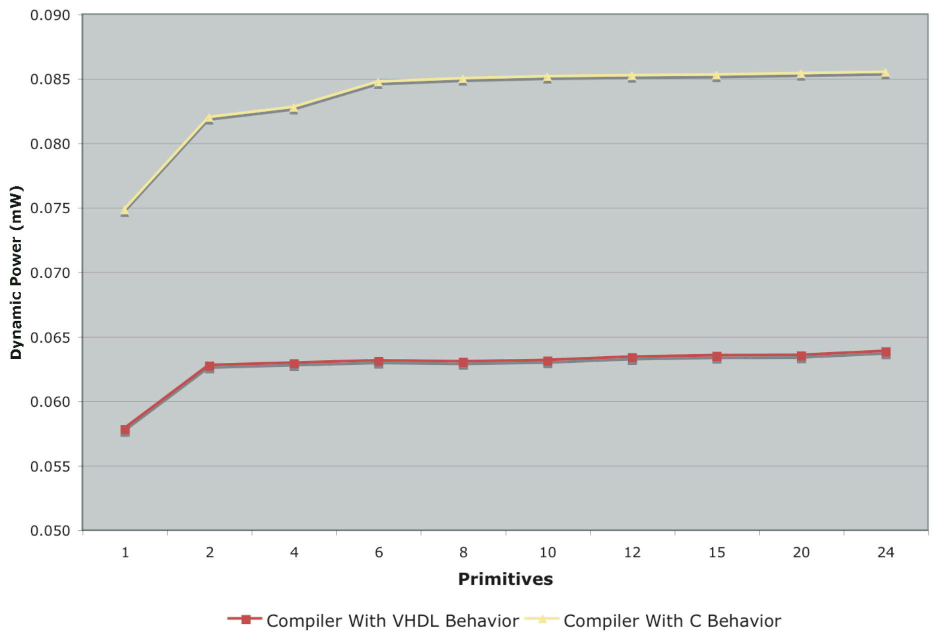


Figure 46: Power Comparison of the Two Hardware RFID Compilers.

6.3.2 Comparison With The ISO 18185 RFID Standard

The ISO 18185 Part 1 standard [12] is an international standard that provides a system for the unique identification and presentation of information about freight container electronic seals. It is used in conjunction with the other parts of ISO 18185 such as, Part 4 that specifies data protection and Part 7 that specifies the physical layer protocol.

The electronic seal mandatory data includes `seal id`, `seal status`, battery status, details on the sealing and opening times, and protocol information. The `seal id` is a combination of the `tag manufacturer id` and the `tag id` and is used to uniquely identify the seal. It is permanently programmed into the seal during manufacturing. The `seal status` indicates the open, closed or sealed state of the seal.

Figure 47 shows the interrogator to tag command / response formats for the *Sleep All But* and *Get Seal Model* commands. The command contains fields such as a `protocol` to identify the data link layer packet structures, an *opcode* `code` to identify the command and `options` to indicate whether it is a point to point or a broadcast command and whether the command duration fields are present. The command duration fields are specified by the interrogator in point to point commands so that the tag may switch to *sleep* mode after waiting for the described duration. The *Sleep All But* command is a broadcast command. In response to this command, the specified seal remains awake while all the other seals return to *sleep* mode. This command does not require a response back to the interrogator. In the case of *Get Seal Model* command, the tag response includes the *opcode* `code`, the nested `seal status`, and other data fields.

Figure 48 shows an example *RFID macros* file containing the *Sleep All But*, *Get Seal Version*, *Get Seal Model*, and *Collection* primitives. Using the RFID compiler, commands of the ISO 18185 Part 1 standard were implemented with 0.16 μm custom cell-based ASIC hardware. The tool used for synthesis and area estimates is Design Compiler. Power was estimated using PrimePower.

Table 6 shows the total area and the power consumption of ISO 18185 Part 1 seal designs for a 0.16 μm ASIC. Figure 49 shows a comparison of the power consumption of the ISO 18000-7 and the ISO 18185 tag designs. The 18185 design is much smaller than the ISO

Sleep All But Command

Protocol	Options	Interrogator	Code	Length	Manufacturer	Tag	CRC
8 bits	8 bits	16 bits	8 bits	8 bits	16 bits	32 bits	16 bits

Get Seal Model Command

Protocol	Options	Manf.	Tag	Inter.	Code	Min Time	Max Time	Len	CRC
8 bits	8 bits	16 bits	32 bits	16 bits	8 bits	16 bits	16 bits	8 bits	16 bits

Get Seal Model Response

Protocol	Status	Length	Interrogator	Manufacturer	Tag	Code	Model	CRC
8 bits	16 bits	8 bits	16 bit	16 bits	32 bits	8 bits	8 bits	16 bits

Status

Mode	State	Reserved	Acknowledge	Reserved	Type	Reserved	Reserved	Battery
4 bits	2 bits	1 bit	1 bit	2 bits	3 bits	1 bit	1 bit	1 bit

Figure 47: Example command/response formats from ISO 18185 Part 1.

18000 Part 7 tag design. From Figure 49, we can see that the power consumption is also correspondingly lower. This could be in part because the commands in ISO 18185 Part 1 do not incorporate security mechanisms such as passwords. For example, ISO 18000 Part 7 provides a password style security mechanism by the *set password*, *set password protect*, and *unlock* commands. There is also an additional layer of privacy introduced by the *user id* field. The logic required to implement the checking of these fields increases the total area and the power consumption of the tag designs.

```
declarations
protcl(8)
options(8)
manuf(16)
tagid(32)
interid(16)
opcode(8)
mindur(16)
maxdur(16)
arglen(8)
crc(16)
status(16)[
  modefield(4)
  state(2)
  reserved(1)
  acknowledge(1)
  reserved2(2)
  sealtype(3)
  reserved3(1)
  reserved4(1)
  battery(1)
]
paclen(8)
wsize(16)
criteria(8)
nodata(0)
version(16)
model(16)

main
sleepbt (22) protcl options interid opcode  arglen  manuf  tagid    crc
             nodata

sealver (12) protcl options manuf   tagid   interid opcode mindur   maxdur  arglen crc
             protcl status  paclen  interid manuf   tagid  opcode   version crc

sealmd (14)  protcl options manuf   tagid   interid opcode mindur   maxdur  arglen crc
             protcl status  paclen  interid manuf   tagid  opcode   model   crc

collect (16) protcl options interid opcode  arglen  wsize  criteria crc
             nodata
```

Figure 48: Macros specification for *Sleep All But* command from ISO 18185 Part 1.

Table 6: Area and Dynamic Power for implementing the ISO 18185 primitive logic on a 0.16μm ASIC. ASIC area is in $100\mu m^2$. Dynamic Power is in mW. Quiescent Power <0.4μW.

Primitives	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Cells	1904	1940	2015	2120	2278	2348	2415	2447	2491	2526
Area	0.7171	0.7284	0.7398	0.7678	0.7779	0.8125	0.8349	0.8509	0.8664	0.8815
Power	0.0338	0.0371	0.0379	0.0381	0.0382	0.0382	0.0383	0.0384	0.0385	0.0386

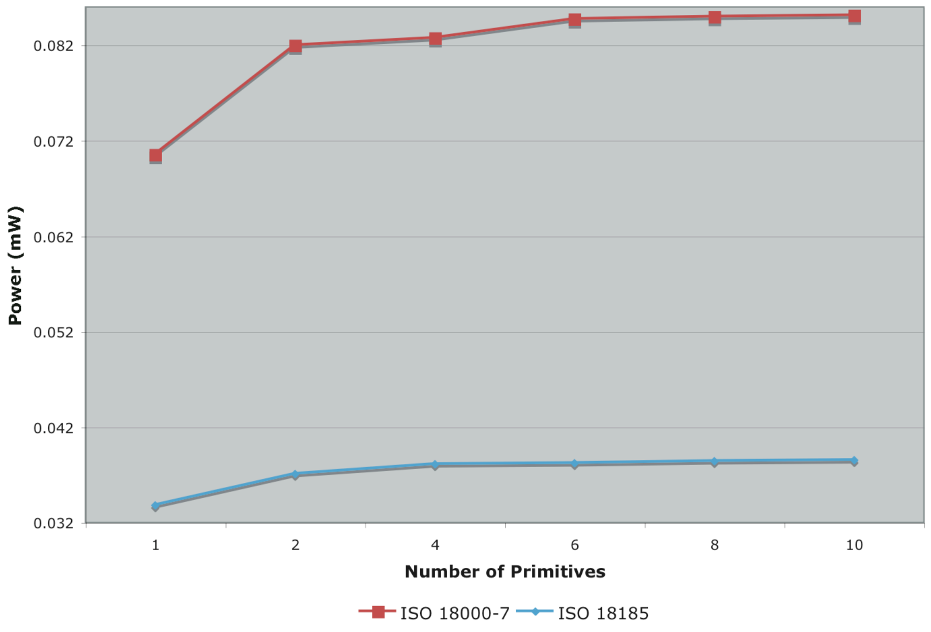


Figure 49: Power consumption comparison for implementing the ISO 18000-7 and ISO 18185 primitive logic on a 0.16μm ASIC.

6.4 THE ISO 18000 PART 6C STANDARD

The ISO 18000 Part 6C UHF standard [13] is becoming a widely accepted standard in Radio Frequency Identification (RFID) applications in supply chain management and is driving development of passive tags. It is a recent amendment to the ISO 18000 Part 6 that describes the RFID air interface for devices operating at 915 MHz and the communications protocols used. Part 6C extends the existing part 6 standard which previously contained type A and B devices with a type C modeled after the EPCGlobal C1 G2 specification.

The communication primitives of ISO 18000 Part 6C standard are significantly different and more complex than the ISO 18000 Part 7 standard. This protocol relies on intermediate storage to retain the state of the system. The size of the command code field is smaller and there are fewer number of commands in the system. This relates to increased complexity of the device in this protocol. As an analogy, the ISO 18000 Part 7 standard is analogous to a Reduced Instruction Set Computer (RISC) style processor while the ISO 18000 Part 6C protocol is analogous to a Complex Instruction Set Computer (CISC) style processor.

The complexity of the Part 6C standard makes the design of these tags extremely time consuming and challenging for reducing power consumption and silicon area. In this chapter, various features of the ISO Part 6C standard are examined and compared to the ISO 18000 Part 7 standard for active tags. The Hardware RFID Compiler is extended to support the design of Part 6C tags.

6.4.1 Description of ISO 18000 Part 6C Standard Interrogator Commands

An interrogator manages tag populations using three basic operations, *select*, *inventory*, and *access*. The selection concept is similar to broadcasts using the *Owner ID* from ISO 18000 Part 7. The *Select* command is applied successively to pick a particular tag population based on user-specific criteria, enabling union, intersection, and negation based tag partitioning. An interrogator begins an "inventory round" by transmitting a *Query* command in one of four sessions. One or more tags may reply to this. The interrogator then detects a single tag reply and requests more information from the tag. The access concept

Table 7: ISO Part 6C Command Codes

Command	Code	Length (bits)	Mandatory?
QueryRep	00	4	Yes
Ack	01	18	Yes
Query	1000	22	Yes
QueryAdjust	1001	9	Yes
Select	1010	>44	Yes
NAK	11000000	8	Yes
ReqRN	11000001	40	Yes
Read	11000010	>57	Yes
Write	11000011	>58	Yes
Kill	11000100	59	Yes
Lock	11000101	60	Yes
Access	11000110	56	No
BlockWrite	11000111	>57	No
BlockErase	11001000	>57	No

is similar to point to point communications in ISO 18000 Part 7. The access operation allows the issuing of commands that read from or write to a tag once the tag is uniquely identified.

Table 7 shows the 18000 Part 6C commands and their command codes. Custom and proprietary commands are supported by the protocol with command codes in the range "E000" - "E1FF".

6.4.2 Tag States and Slot Counter

ISO 18000 Part 6C tags implement features such as accessing and killing passwords, checking the electronic product code (EPC), CRC checking, manipulating the slot counter, pseudo-random number generation, etc. The states and keys of the target device are used to facilitate tag singulation, collision arbitration, security encoding, etc.

Tags implement a 15-bit slot counter, which is used for collision arbitration. The slot counter is loaded with a random value when the tag receives a *Query* command. The *QueryAdjust* and *QueryRep* commands can also modify the values of the slot counter.

Tags have seven states including `ready`, `arbitrate`, `reply`, `acknowledged`, `open`, `secured`, and `killed`. The `ready` state is a holding state for energized tags that are neither killed nor participating in an inventory round. The `arbitrate` state is a holding state for tags that are participating in an inventory round but whose slot counters hold non-zero values. Tags in the `reply` state backscatter the appropriate reply. A tag in the `acknowledged` state transitions to other states based on the command received. Tags in the `open` state can execute all access commands except *Lock*. Tags in the `secured` state can execute all access commands. The *Kill* command puts the tag in the `killed` state, in which it does not respond to any further interrogator commands. This state is not reversible. Figure 50 shows a partial state diagram of the tag. The state transitions introduced by some of the commands are shown.

6.4.3 Tag Memory

ISO 18000 Part 6C tags contain memory segmented into four memory banks. Figure 51 shows the logical memory map of the tag. The memory banks are for user memory, tag identifier (TID) memory, the EPC memory, and a reserved memory. The user memory can be used for user specific data storage. The TID memory contains the TID and may contain other vendor or tag specific data. The EPC memory contains a CRC-16, the protocol control bits (PC), and the EPC that identifies the type of object to which the tag will be attached. The reserved memory contains the kill and access passwords. The access password is a 32-bit value that must be issued to put the tag in the secured state. The kill password is a 32-bit value that is used to permanently disable the tag. The interrogator uses the *Write*, *BlockWrite*, and *BlockErase* commands to edit the memory.

6.4.4 Security Features

The interrogator can lock or unlock each individual area of memory. This includes the access password, kill password, the EPC memory bank, the TID memory bank and the user memory bank. Tags must be in their secure state to use the *lock* command. Locking read and write protects the passwords and write protects the other memory banks. The interrogator can also *permalock* the lock status for a password or memory bank so that it is

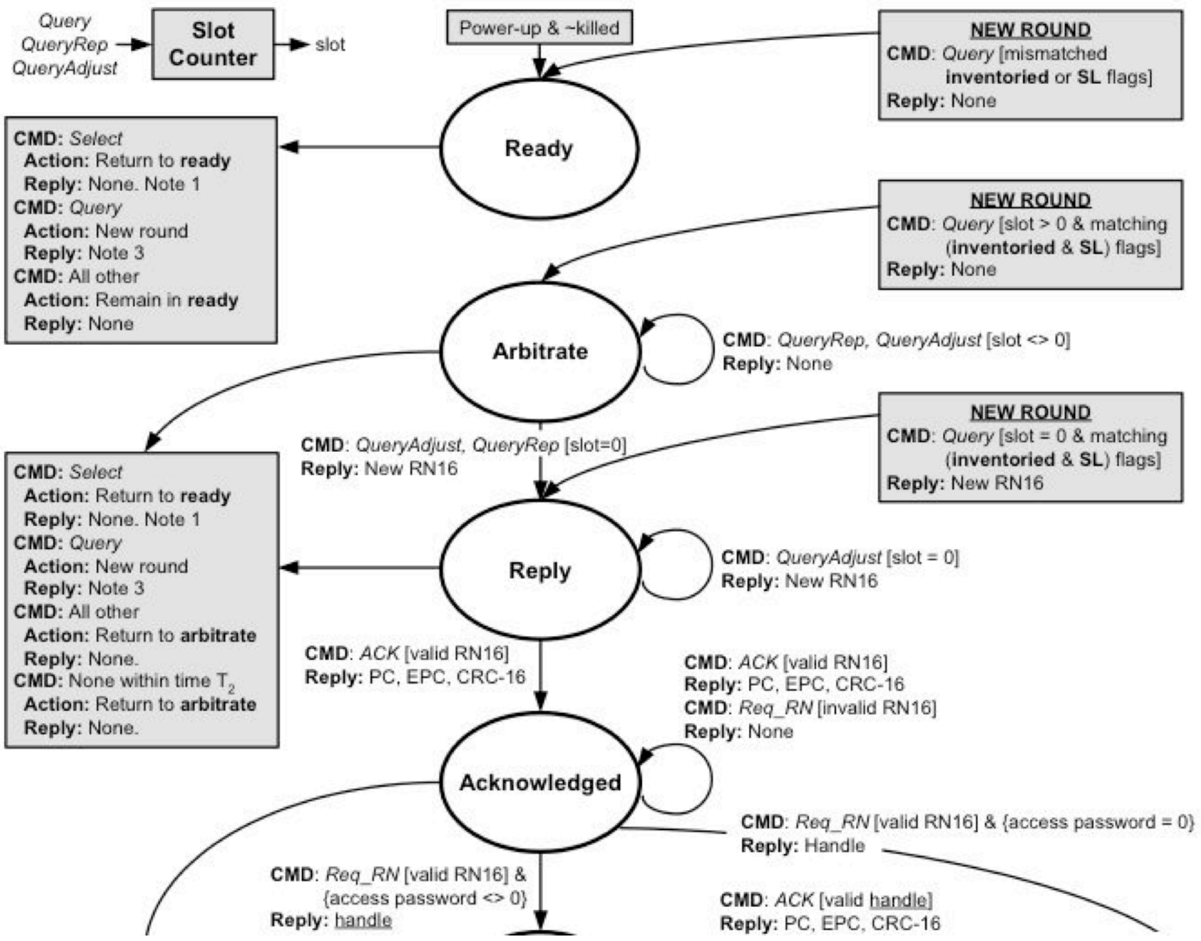


Figure 50: Partial state diagram of ISO Part 6C tag from the EPCGlobal Class 1 Generation 2 document [4].

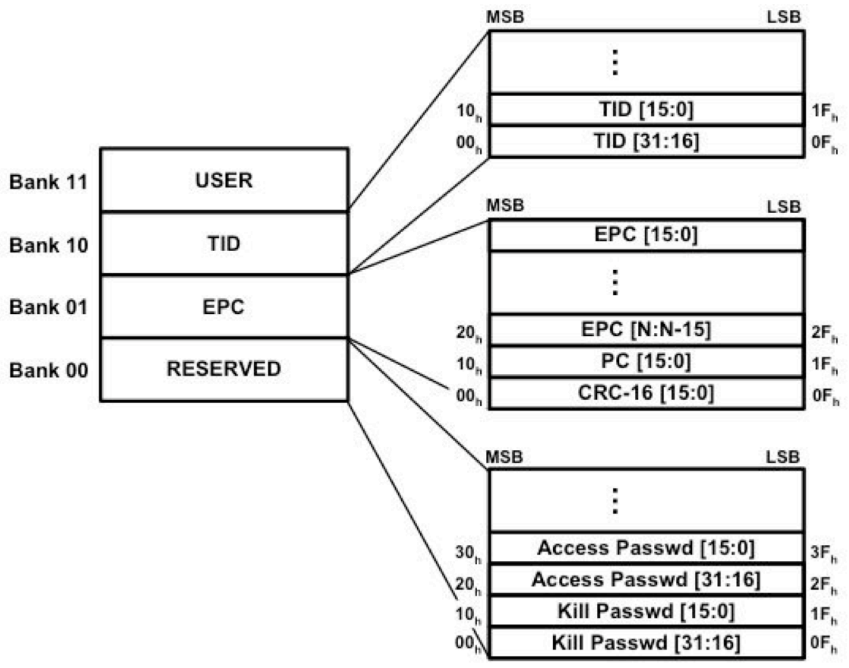


Figure 51: Logical Memory Map of ISO Part 6C Tag [4].

unchangeable. Permalock bits, once asserted, cannot be deasserted. This prevents the memory from being altered maliciously. Another security feature in the 18000 Part 6C standard is the *Kill* command. It can permanently deactivate the tag when the interrogator issues the correct kill password. After this is executed, the tag no longer responds to interrogator commands.

Thus, the ISO 18000 Part 7 and Part 6C protocols represent opposite ends of the complexity spectrum for tag implementations. As shown in the protocol overviews from Sections 4.1 and 6.4, the ISO 18000 Part 7 standard is analogous to a RISC style processor and the ISO 18000 Part 6C standard is analogous to a CISC style processor. RISC processors are based on a small number of simple instructions, but require a large instruction count for an application. CISC processors have a number of complex instructions that allow an application to require a much lower instruction count. Typically, embedded processors that target reduced power as a metric are RISC style processors with the prime example of the ARM processor family. General purpose processors, particularly for desktop computers

Query Command

QCmd	**DR**	**M**	**TRext**	**Sel**	**Session**	**Target**	**Q**	**CRC-5**
4 bits	1 bit	2 bits	1 bit	2 bits	2 bits	1 bit	4 bits	5 bits

Response

RN16
16 bits

Figure 52: *Query* command and response format from ISO Part 6C.

where power is only a concern for heat dissipation, the processor architectures are more CISC with vector and very long instruction word (VLIW) processing engines to augment the processor.

6.5 IMPLEMENTING ISO 18000 PART 6C

The commands or primitives issued by the interrogator request that the tag perform a set of actions. The specifications of these commands widely vary from ISO Part 7 to Part 6C. The RFID compiler that targets a hardware-based controller, which was described in Chapter 6, has been used to implement the ISO Part 6C primitives.

Figure 52 shows the fields for the first communication step of the ISO Part 6C *Query* command, which achieves similar functionality as the *Collection* command. However, the *Query* command has several subsequent communications as indicated by Figure 53. To automate the generation of the tag controller for a prototype implementation, the *primitives* are implemented as *RFID macros* as shown in Figure 54. For comparison, the fields required for executing the *Collection* command of the Part 7 standard are shown in Figure 14.

The command code field of Part 6C commands can have variable bit widths. This decimal value is also included in the macros specification. For example, in Figure 54, this is indicated by “4”. The decimal value of the command code for *Query* is indicated by “8”. The template generated for the *Query* command is shown in Figure 55. The behavior for the *Query* command is shown in Figure 56.

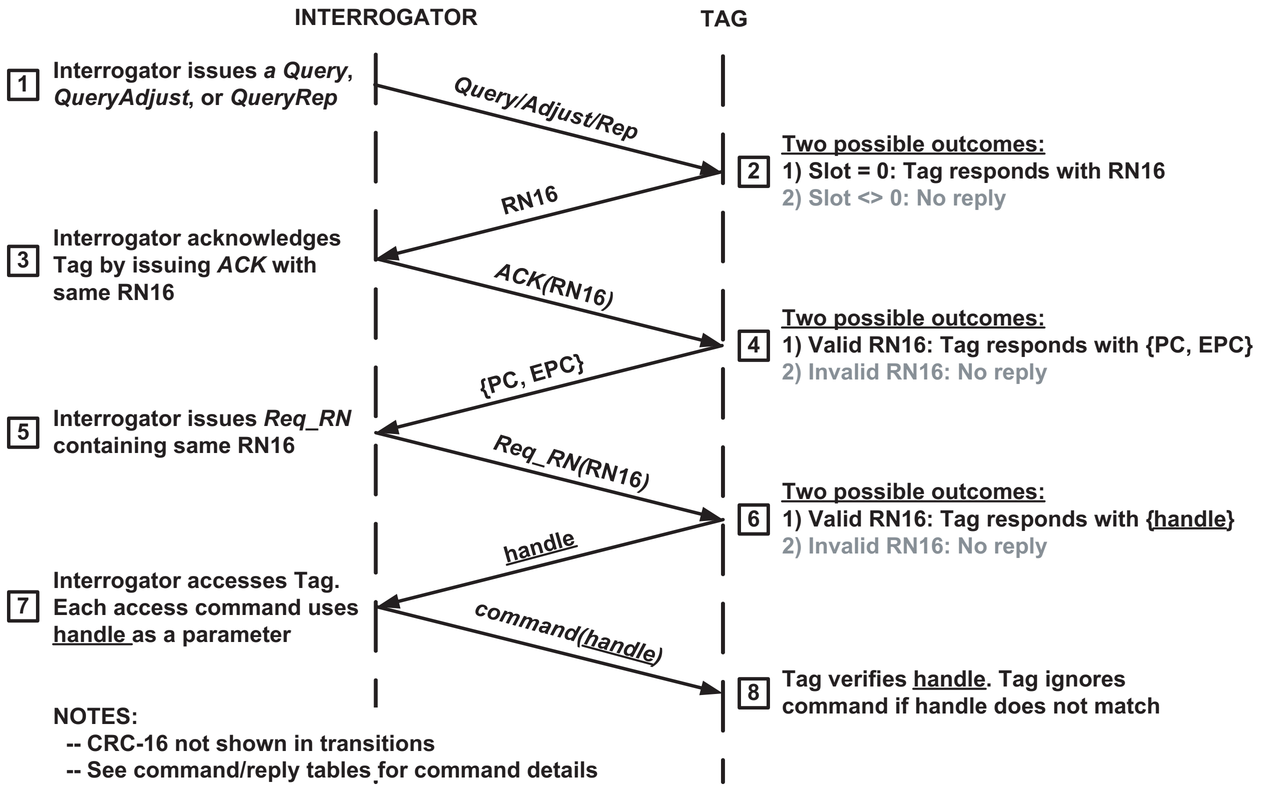

INTERROGATOR
TAG
1 Interrogator issues a Query, QueryAdjust, or QueryRep
Query/Adjust/Rep
2 Two possible outcomes:
1) Slot = 0: Tag responds with RN16
2) Slot <> 0: No reply
RN16
3 Interrogator acknowledges Tag by issuing ACK with same RN16
ACK(RN16)
4 Two possible outcomes:
1) Valid RN16: Tag responds with {PC, EPC}
2) Invalid RN16: No reply
{PC, EPC}
5 Interrogator issues Req_RN containing same RN16
Req_RN(RN16)
6 Two possible outcomes:
1) Valid RN16: Tag responds with {handle}
2) Invalid RN16: No reply
handle
7 Interrogator accesses Tag. Each access command uses handle as a parameter
command(handle)
8 Tag verifies handle. Tag ignores command if handle does not match
NOTES:
-- CRC-16 not shown in transitions
-- See command/reply tables for command details

Figure 53: Communications required as part of the 18000 Part 6C *Query* command [4].

```
declarations
DR(1)
M(2)
TRext(1)
Sel(2)
Session(2)
Target(1)
Q(4)
CRC-5(5)
RN16(16)
RN(16)
PC(16)
EPC(16)
CRC-16(16)
MemBank(2)
WordPtr(8)
Data(16)
Header(1)

main
query(4,4)     DR       M        TRext   Sel  Session  Target  Q  CRC-5
               RN16

ack(2,1)       RN
               PC       EPC      CRC-16

req_rn(8,193)  RN       CRC-16
               RN       CRC-16
```

Figure 54: Macros specification file for commands required as part of the 18000 Part 6C *Query* command.

```
RN16 =
inventoryFlag =
current_state =
...
```

Figure 55: Template generated for *Query* command.

```
if (current_state != KILLED) {
  if ((current_state == ACKNOWLEDGED) || (current_state == OPEN)
      || (current_state == SECURED)) && ((sel_var == sel)
      && (target_var == target))) {
    if (session == last_session) {
      if (inventory_flag == 'A')
        inventory_flag = 'B';
      else if (inventory_flag == 'B')
        inventory_flag = 'A';
    } else {
      slot_counter = rand((1<<Q) - 1);
      if (slot_counter != 0)
        current_state = ARBITRATE;
      else
        current_state = REPLY;
      if (current_state == REPLY)
        RN16 = slot_counter;
    }
  }
}
```

Figure 56: Tag behavior for *Query* command.

6.5.1 Extensions to The Hardware RFID Compiler For ISO 18000-6C

The Part 6C Tag's behavior and the next state logic for each command is different for each of the 7 states that it can be in. Apart from this state-dependent logic for each command, there is also logic that is performed state-independently. For the example, when the *Query* command is received, the slot counter is loaded with a pseudo-random number, if the Tag is in any state other than *killed*. The Compiler has been modified so that the state variables, global keys, etc can be declared in the macros file, and they are generated as global VHDL signals (or global variables in C). The primitive components are then able to share values of state variables and keys.

The command code field of Part 6C commands can have variable bit widths, which is input in the macros file, as shown in Figure 54. The Compiler has been modified to parse the width of the command code and use it while extracting the opcode and remaining fields in the command packet.

Implementation of Random Number Generation: The Part 6C standard specifies that tags should implement a random or pseudo-random number generator. There are a number of ways to generate pseudo-random numbers in hardware. The linear feedback shift register (LFSR) is the most commonly used. The LFSR is a shift register which, using feedback, modifies itself on each rising edge of the clock. The feedback causes the value on the shift register to cycle through a set of unique values. One of the ways to implement an LFSR is using the Fibonacci implementation, where the outputs from some of the registers are Exclusive OR-ed together and fed back to the input of the shift register. For example, Figure 57 shows a 3-bit Fibonacci LFSR. When the shift register is loaded with a non-zero seed value and then clocked, the output from the LFSR (Q2) will be a pseudo-random binary sequence. For example, Table 8 shows the sequence that is produced from the circuit in Figure 57 when the seed is 111.

The length of the sequence depends on the length of the shift register and the number and position of the feedback taps. This is expressed using a polynomial. By selecting the appropriate polynomial, the maximal length sequence for the given LFSR size can be obtained. For a 16-bit LFSR, by using outputs from registers 16, 15, 13 and 4 in a Fibonacci implementation, the corresponding maximal sequence length of 65535 can be obtained [39]. This has been implemented in the RFID Compiler. Different LFSR implementations were considered, but they were found to be larger and more power consuming compared to the LFSR shown in Figure 58. It was the optimal choice in terms of area and power and it also meets Part 6C's specification of the probability of the 16-bit random number (Section 6.3.2.5 of [13]). According to the specification, the probability that any RN16 drawn from the random number generator has value RN16 = j, for any j, shall be bounded by $0.8 / 2^{16} < P(\text{RN16} = j) < 1.25 / 2^{16}$. The probability of our 16-bit LFSR is $1 / (2^{16} - 1)$ and is $1.53 \cdot 10^{-05}$.

Thus, a 16-bit random number is generated by the LFSR. The standard specifies that tags should be able to extract Q-bit subsets from the RN16 to preload the tag's slot counter. The range of the random number is to be limited to $(0, 2^{Q-1})$. For this a simple 'AND' logic is implemented. For example, consider the 16-bit random number, 0x1A0D, and a Q value of 3. The corresponding value of 2^{Q-1} is 7. The random number is AND-ed with 7, resulting in 5, which is in the range (0, 7).

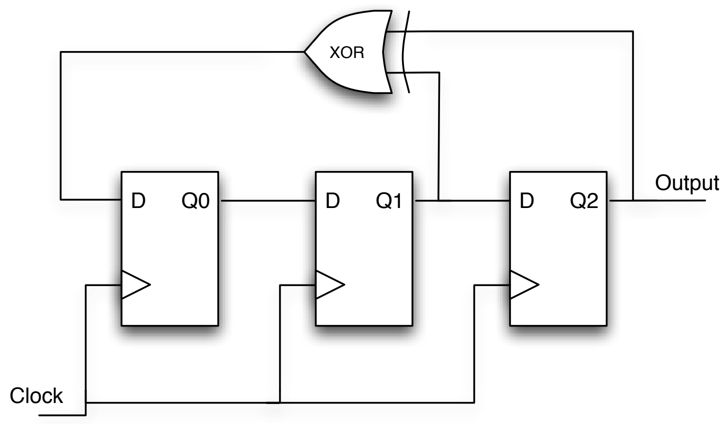


Figure 57: A 3-bit Fibonacci LFSR.

Table 8: LFSR Sequence For The Seed '111'.

Q2	**Q1**	**Q0**
1	1	1
1	1	0
1	0	0
0	0	1
0	1	0
1	0	1
0	1	1
1	1	1

6.5.2 Comparison with ISO 18000 Part 7

To understand and compare the complexity of different standards such as the ISO 18000 Part 7 and Part 6C, the RFID compiler has been used to implement commands from both standards. A representative command, *Query*, was selected from Part 6C and has been implemented in hardware and synthesized for an ASIC. Similarly, the *Collection* command, which realizes similar functionality from Part 7 standard, was implemented in hardware and targeted to the same ASIC process. Table 9 shows the power and area results for implementing these two commands. The *Query* command is larger and requires more power than the *Collection* command. We also compared the *Query* command and the *Collection* command with nine additional primitives from the ISO 18000 Part 7 standard. As shown in Table 9, the *Query* command is still larger and consumes more power than these.

6.5.3 Custom Hardware-based Tag

Using the RFID compiler, up to five inventory commands of the ISO 18000 Part 6C standard were implemented with 0.16 μm custom cell-based ASIC hardware. To evaluate the effectiveness of the automated approach in providing a rapid prototype and accurate estimate of resource requirements of the RFID system, the areas of the automated tag designs generated by the RFID Compiler have been compared to the areas of our own manual tag designs for the above targets. The tool used for estimating area is Design Compiler.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;

entity gen2lfsr is
  port(
    clk : in STD_LOGIC;
    rst : in STD_LOGIC;
    generateQ : in STD_LOGIC;
    Q   : out UNSIGNED(16 downto 1)
  );
end gen2lfsr;

architecture a of gen2lfsr is

  signal lfsr : UNSIGNED(16 downto 0);
  signal load_v : UNSIGNED(16 downto 1);
  constant lfsr_cmp : UNSIGNED(16 downto 0) := (others => '0');
  signal load : STD_LOGIC;

begin
  Q <= lfsr(16 downto 1);
  load_v(16 downto 2) <= (others => '0');
  load_v(1) <= '1';
  load <= '1' when (lfsr = lfsr_cmp) else '0';
  lfsr(0) <= '0';

  process(clk, rst, load_v, load, generateQ)
  begin

    if (rst = '1') then
      lfsr(16 downto 1) <= (others => '0');
    elsif (rising_edge(clk)) then
      if (generateQ = '1') then
        if (load = '1') then
          lfsr(16 downto 1) <= load_v;
        else
          lfsr(1) <= (lfsr(16) xor lfsr(15)) xor (lfsr(13) xor lfsr(4));
          lfsr(16 downto 2) <= lfsr(15 downto 1);
        end if;
      end if;
    end if;
  end process;
 end a;
```

Figure 58: VHDL for 16-bit LFSR.

Table 9: Power and energy results for implementing *Query*, *Collection* and 10 ISO Part 7 primitives (inclusive of *Collection*) as a 0.16μm ASIC. ASIC area is in $100\mu m^2$. Dynamic power is in mW. Quiescent Power <0.4μW.

Primitives	**Dynamic Power (mW)**	**Area** ($100\mu m^2$)
Query	0.06752	1.1642
Collection	0.06308	1.0944
10 primitives	0.06495	1.1232

Table 10 shows the total area of ISO 18000 Part 6C tag designs for a 0.16um ASIC. For the ASIC implementation of five primitives, there is a very large increase in area of up to 66.56%. This can be compared to the nominal increase in area of 9.37% in the case of the compiler-generated ISO 18000 Part 7 tag with 24 primitives, shown in Section 6.3. The reason for this increase is explained below.

Consider the C behavior for the *Query* command, shown in Figure 56. It contains multiple *if-then-else* conditional constructs to check the various states, sessions, variables, etc. *If-then-else* constructs can be implemented in hardware using a multiplexer acting as a binary switch to predicated output datapaths. In the CDFG representation of an *if-then-else* statement, control flow creates basic block boundaries with control flow edges. Using hardware predication, these control flow dependencies are converted into data flow dependencies. Each symbol defined in either or both of the *then* and the *else* basic blocks is predicated by inserting a multiplexer.

The ISO 18000 Part 6C protocol is complex and uses multiple states and keys. Due to the large number of symbols and multiple levels of *if-then-else* statements in the C code, many multiplexers may have been instantiated in the hardware. In the ASIC design flow, design compiler automatically performs resource sharing if no timing constraints are violated. The tool identifies sharable resources as the ones used in the same *case* statement or *if* statement branches, which is not the case for most of the multiplexers generated in the automated VHDL design. Thus, for these designs, resource sharing has not occurred, thereby leading to greatly increased area. Manual design compiler controls can be potentially used for

Table 10: Area for implementing the original compiler-generated Part 6C primitive logic on a 0.16μm ASIC. ASIC area is in $100\mu m^2$.

Primitives	**1**	**2**	**3**	**4**	**5**
Manual	1.1642	1.1933	1.2288	1.2313	1.3212
Automated	1.3817	1.5883	1.7913	1.8760	2.2006
% increase with automation	18.68%	33.10%	45.78%	52.36%	66.56%

increased sharing to reduce the area in the ASIC designs. Further, all the variables in the automatically generated VHDL components are 32 bits wide, since the SuperCISC compiler has no knowledge of their actual width. Most of the variables are originally much smaller, for example, the *current_state* variable in Figure 56 is only 3 bits wide. This explains the smaller area of the manual implementations. This motivates the need for improving the compiler synthesis.

6.6 COMPILER OPTIMIZATIONS

Area optimizations have been built in to the RFID Compiler. A precision pass updates the sizes of the CDFG input and output nodes, followed by a conversion pass that examines each operation node and reduces its size to the minimum bit width calculated based on its inputs. Figure 59 shows an outline of the precision pass that updates the sizes of CDFG nodes. An example precision file used in the design of a Part 6C tag with five commands is shown in Figure 60. The width of most of the variables is between 1 and 3 bits.

The result of these optimizations is a substantial reduction in the area of the original design, of up to 33.63%, as seen in Figure 61 and Table 11.

```
void Precision::do_file_set_block(FileSetBlock *fsb) {
  // iterate through the files that were loaded
  for ( all file set blocks) {
    FileBlock *fb = fsb->get_file_block(i);
    // if there are cdfgs generated
    cdfgAn = to<cdfgAnnote>(fb->peek_annote("cdfg_annote"));
    // get the list of cdfgs
    list<cdfg*> *cdfg_list = cdfgAn->get_cdfg_list();
    for ( all cdfgs ) {
      if(the_cdfg->is_active()) {
        // function reads the precision file and
        // stores each node and precision info
        // in a map data structure
        fillPrecisionArray();
        for( all basic blocks ) {
          baselist = basicblocks->get_IO_nodes(the_cdfg->is_live());
          if(baselist != NULL) {
            for( all cdfg io nodes ) {
              // cast to cdfg base node
              cdfg_base *cb = *CBiter;
              if (cb != NULL) {
                // lookup node's precision value from map using name
                int intp = GetPrec(LString(cb->get_string()));
                // set new precision
                cb->set_prec(intp);
                cb->set_sign(false);
              }
            }
          }
        }
      }
    }
  }
}
```

Figure 59: Outline of Precision Pass That Updates Sizes of CDFG Nodes.

```
current_state  3
next_state  3
inventory_flag  1
last_session  2
session  2
target_var  1
target  1
sel_var  2
sel  2
updown  3
Q  4
RN16  16
```

Figure 60: Example precision file.

6.7 RESULTS

Using the final version of the hardware RFID compiler, up to five inventory commands of the ISO 18000 Part 6C standard were implemented with 0.16 μm custom cell-based ASIC hardware and a Spartan 3 FPGA. The tag controller designs were implemented by adding commands incrementally to the design. The commands implemented are *Query*, *QueryAdjust*, *QueryRep*, *Nak*, and *Ack*. The areas of the automated tag designs generated by the RFID Compiler have been compared to the areas of manual tag designs for the above targets. The tools used for estimating area are Design Compiler and Precision Synthesis for ASIC and FPGA targets, respectively.

Tables 11 and 12 show the total area and power consumption of ISO 18000 Part 6C tag designs for ASICs. Table 13 shows the resource utilization of Part 6C tag designs for a Spartan 3 FPGA. For the ASIC implementation of five primitives, there is a nominal increase in area of up to 10.55%. We note that there is a trend that as primitives are added the area increase percentage rises. This is in part due to the hardware predication in the SuperCISC compiler and how design compiler does resource sharing as previously explained in Section 6.5.3. Design compiler does allow resource sharing through use of specialized controls, which provide an opportunity to reduce this overhead.

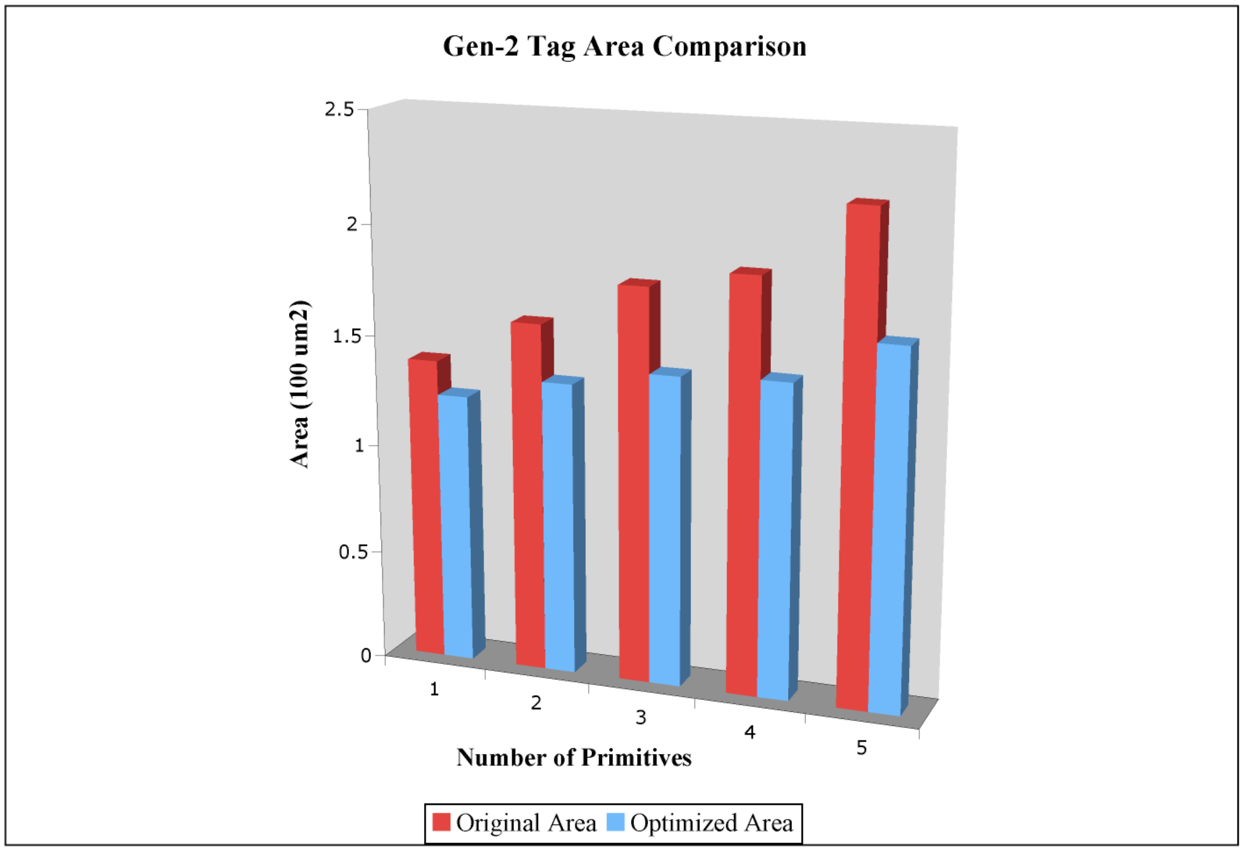


Figure 61: Area for implementing the Part 6C primitive logic on a 0.16μm ASIC with and without optimizations.

Table 11: Area for implementing the fully optimized Part 6C Inventory primitive logic on a 0.16μm ASIC. ASIC area is in $100\mu m^2$.

Primitives	**1**	**2**	**3**	**4**	**5**
Manual	1.1642	1.1933	1.2288	1.2313	1.3212
Automated	1.1326	1.2159	1.2842	1.2942	1.4606
% increase with automation	-2.71%	1.89%	4.51%	5.11%	10.55%
% improvement as a result of optimizations	18.03%	23.45%	28.31%	31.01%	33.63%

It can be seen that this area increase did not occur with other synthesis tools for FPGAs. The FPGA resource utilization is almost the same for both the approaches and actually improves slightly with the automated approach. The utilization of CLB slices and latches consistently improved with the use of design automation, for the tag with up to five primitives. While the utilization of global buffers and IOs remained constant, the utilization of function generators was variable.

Table 14 shows the dynamic power and total area of more ISO 18000 Part 6C tag designs for a 0.16um ASIC. The commands that have been added are *Select*, *Req_RN*, *Read* and *Write*. These results also confirm the fact that the ISO 18000 Part 6C tag designs are significantly larger and more complex than tag designs of other standards. However, with the use of the design automation tool, tag implementations that are very similar in area and power to corresponding manual designs can be achieved in less time.

Table 12: Dynamic power for implementing the fully optimized Part 6C Inventory primitive logic on a 0.16μm ASIC. Dynamic power is in mW. Quiescent Power <0.4μW.

Primitives	**1**	**2**	**3**	**4**	**5**
Manual	0.0640	0.0650	0.0657	0.0659	0.0666
Automated	0.0649	0.0657	0.0670	0.0676	0.0682
% increase with automation	1.34%	1.05%	1.98%	2.52%	2.40%

Table 13: Resource utilization for implementing the Part 6C Inventory primitive logic on a Spartan 3 FPGA.

Primitives	**1**	**2**	**3**	**4**	**5**
IOs					
Manual	419	419	419	419	419
Automated	419	419	419	419	419
% increase with automation	0%	0%	0%	0%	0%
Global Buffers					
Manual	2	2	2	2	2
Automated	2	2	2	2	2
% increase with automation	0%	0%	0%	0%	0%
Function Generators					
Manual	720	757	787	789	817
Automated	713	742	814	814	825
% increase with automation	-0.97%	-1.98%	3.43%	3.17%	0.98%
CLB Slices					
Manual	569	572	580	580	588
Automated	559	563	571	571	571
% increase with automation	-1.76%	-1.57%	-1.55%	-1.55%	-2.89%
Dffs or Latches					
Manual	1138	1143	1159	1159	1176
Automated	1118	1125	1141	1141	1141
% increase with automation	-1.76%	-1.57%	-1.55%	-1.55%	-2.98%

Table 14: Area and Dynamic power for implementing the fully optimized Part 6C Access primitive logic on a 0.16μm ASIC. ASIC area is in $100\mu m^2$. Dynamic power is in mW. Quiescent Power <0.4μW.

Primitives	**6**	**7**	**8**	**9**
Area	1.5114	1.5630	1.6136	1.6566
Cells	4281	4463	4593	4756
Power	0.06824	0.06829	0.06840	0.06898

7.0 TECHNIQUES FOR OPTIMIZING THE TAG POWER CONSUMPTION

With the increasing power density of modern circuits as the number of transistors per chip scales, power efficiency has increased in importance. Power consumption has become important in servers and portable devices like laptops. Also, in embedded computing, power efficiency has long been and remains the primary design goal next to performance.

However, for RFID tags, power consumption has been held as the secondary metric. Typically, the biggest metric of concern for the silicon controller devices for the tag has been cost. In chip fabrication terms, the cost of the device directly relates to the CMOS process chosen for implementation and the area of the device produced. Older CMOS processes such as 0.18 μm to 0.35 μm are targeted for RFID as they are relatively cheap to produce than the newer 65 nm - 90 nm processes. The design goal is to create a device with the smallest area using one of these older processes.

RFID protocols are typically designed without taking into account many of the impacts of their final implementation. For example, the design of a protocol can significantly impact the complexity of the protocol realization requiring additional area and cost in the final implementation. Additionally, this complexity can impact the power consumption. Even decisions about primitive opcode encoding can significantly impact power consumption while only minimally impacting area. With existing design flows, to gain an accurate estimate of power consumed for a protocol implementation, the protocol must be designed, tested for correctness, implemented in hardware, and finally studied for power. This process can take months or years of engineering effort to complete. RFID companies typically do not have this type of man power available to dedicate for this purpose.

Power consumption is critical in both passive and active devices. For an active device, the amount of power/energy consumption required by the tag dictates the lifetime that a tag may operate. For a passive device, the range and complexity of computation for features such as added security capability or access to sensors are directly impacted by power usage. Since the amount of available energy is limited, it has to be budgeted wisely by the tag.

To enable low-power design, a power macro-modeling technique that works in concert with the RFID design automation flow and calculates the power estimate of the tag so designed is presented in this dissertation. While this technique is targeted towards hardware-based tags, an application specific instruction processor (ASIP) is studied for possible reduction of power in the design of microprocessor-based tags.

7.1 POWER MACROMODELING

7.1.1 Motivation

As shown in this dissertation, the RFID compiler, as illustrated in Figure 62, allows the RFID system designer to design and implement new RFID protocols in a matter of hours. A team of design engineers without this tool would require months or longer. By requiring the team to examine the power and area impacts of their completed designs to optimize the tag would require additional time and effort.

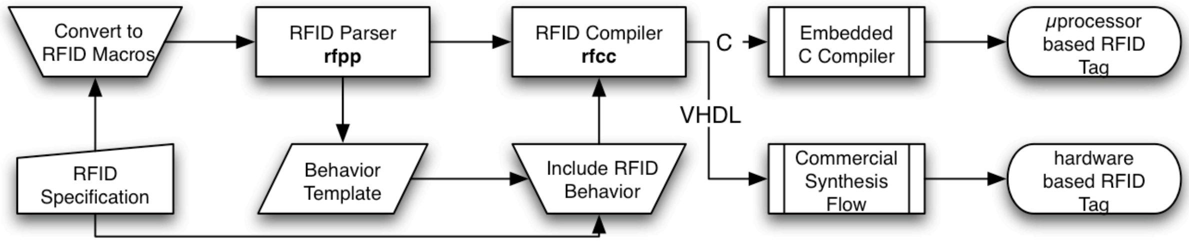


Figure 62: RFID high-level specification methodology and compilation flow.

However, the generation of the performance and area details of the design requires specialized ASIC synthesis tools such as Synopsys Design Compiler which must be manually tuned to achieve good results. Performance and area may be estimated at this level but require additional computer aided design effort such as placement, routing, and design rule checking with tools like Cadence SoC Encounter to get a more accurate result, requiring additional time and effort. Achieving power consumption statistics requires an additional level of effort by simulating the design and putting it through additional power estimation tools such as Synopsys Nanosim, or HSPICE.

Hardware system designers have come up with diverse approaches for energy/power reduction at all levels of abstraction starting from the physical level up to the system level. Experience shows that a high level method may have a larger impact since the degree of freedom is still very high. In order to conduct efficient system-level optimizations, high-level power modeling tools are required.

In the following subsections, a tool is described based on a power macromodeling technique that calculates a power estimate at a high level during the design automation process of the RFID compiler. The tool generates a behavioral representation of the hardware that generates a custom simulator for the controller design generated by the RFID compiler. Through access to a pre-profiled library of blocks in the target CMOS process, the power consumption can be estimated one hundred times faster than the fastest ASIC power estimation flows. Thus, the user has near instantaneous feedback about the power consumption of their design without detailed knowledge of ASIC synthesis and power estimation flows.

7.1.2 Power Macromodeling Framework

Power macromodeling was originally proposed by Gupta and Najm as a fast power estimation technique that considers the impact of different input combinations on the circuit [40]. This technique has been shown to be far more accurate than static power estimation methods that do not consider design input values. Power macromodeling has been proposed for a variety of purposes including high-level synthesis of circuits for minimal power consumption [41, 42]. Power macromodeling discretizes the components used to build up the circuit into functional blocks that can be implemented and analyzed for their power consumption based on different input stimuli. These results are then compiled into a lookup table of power consumption based on different characteristics of the input stimuli. During a behavioral simulation of the system, rather than computing the power consumption of the block based on the simulated inputs, a table look-up is performed to determine the power value at a significant savings in time and effort.

An overview of our power macromodeling flow used in our RFID compilation flow is shown in Figure 63. The SystemC generation occurs just prior to VHDL generation in Figure 62. The RFID compiler reads the RFID protocol description, including the RFID macros and their corresponding behavioral in C and automatically generates a system level simulator for the tag in the SystemC language.

SystemC is a hardware description language built using C++ libraries that include facilities for discrete event simulation, concurrency, fixed width bit vectors, and block-based design, amongst others. As it is at heart a C++ program, a SystemC description of the hardware design is compiled into an executable custom simulator for that particular hardware design [43]. SystemC is particularly appropriate for power macromodeling as the power table and estimation behavior can be seamlessly integrated into the simulator through C++ code. Additionally, the final simulator is typically fast as the simulator is a compiled binary rather than an interpreted simulation of other hardware description languages.

After the SystemC simulator is generated, it is compiled into a binary using a software compiler. Probabilistic input test vectors are then automatically generated for use in simulation. The compilation flow executes the simulation and generates annotated trace files with information about all the functional units in the design. These functional units have been pre-profiled for power based on input parameters described in more detail in Section 7.1.2.1. The power estimation of each RFID primitive and the overall tag design is calculated by combining the trace information with the profiles to determine each units power, which is aggregated. The resultant power estimates correspond to the actual activity of the tag behavior in hardware.

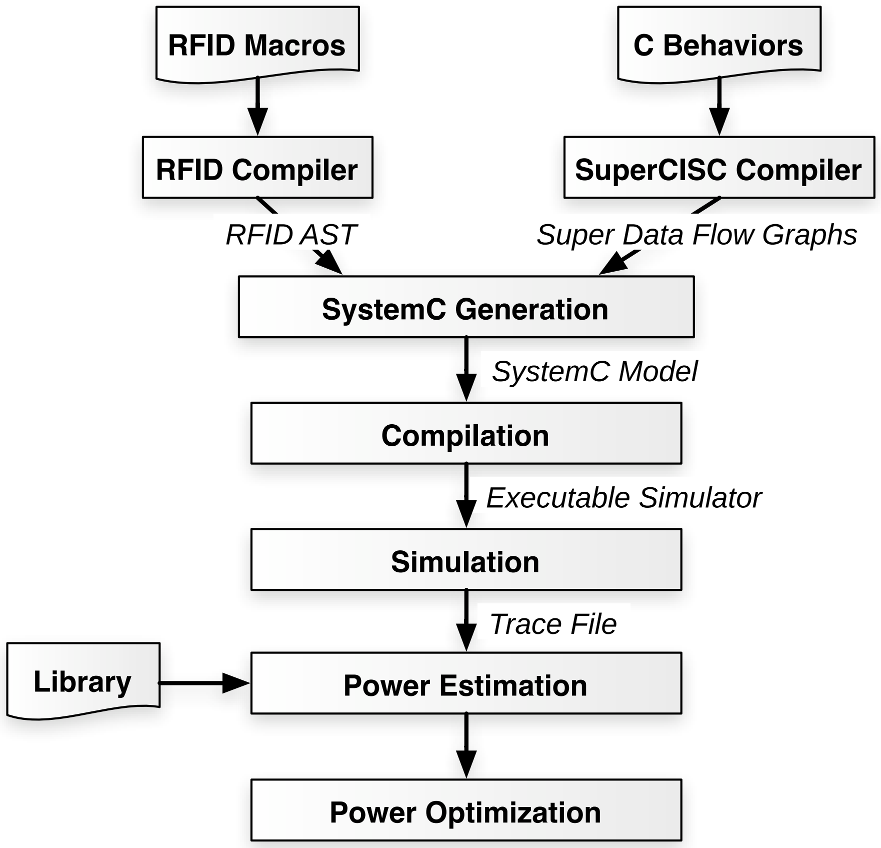


Figure 63: Power macromodeling flow. Prior to VHDL code generation in Figure 62, this flow generates a SystemC model to allow power analysis. The user changes the specification until an acceptable power result is achieved, and then Figure 62 resumes at VHDL generation.

7.1.2.1 Library of Power Profiles It is often assumed that the only statistic for that impacts the dynamic power consumption is the density of bit transitions between input vectors. However, it has been suggested that three parameters of input signals are important to accurately estimate power dissipated in digital circuits [40, 44, 45]. They are: average input signal probability, p, the aforementioned average transition density, d, and spatial correlation, s. Transition density represents the frequency of bit changes between two or more values in sequence. Signal probability describes the number of '1's to appear within a value. Spatial correlation describes the likelihood for '1's and '0's to appear in groups within the value.

All the types of functional units used by the RFID compiler for hardware generation, such as addition, equivalence, multiplexers, etc., have been power profiled with different values of p, d, and s. These have been used to construct a library of power profiles[1]. We used the Markov chain-based sequence generator described in [46] that converts the probabilistic p, d, and s values into an actual sequence of test vectors for use in simulation. Measurements were taken at 0.1 intervals ranging from 0.05 to 0.95 in each probabilistic dimension. The functional units are synthesized using 0.16μm Oki cell-based ASIC technology. The synthesis was executed with Synopsys Design Compiler and the power was estimated using Synopsys PrimePower.

The power consumption for functional units such as adder, multiplier, AND operation and XOR operation profiled as described above is displayed in Figure 64. This chart plots power versus p, d, and s. Power is indicated as a color between black and white where solid black represents the least power consumed by the device and solid white indicates the most power consumed by the device. From Figure 64, it can be seen that the most power is consumed by the adder when spatial correlation is low and transition density is high. This shows that using transition density, d, alone for dynamic power optimization can be insufficient, although it is often considered the only metric of interest.

[1]The power profiles were originally published in [38].

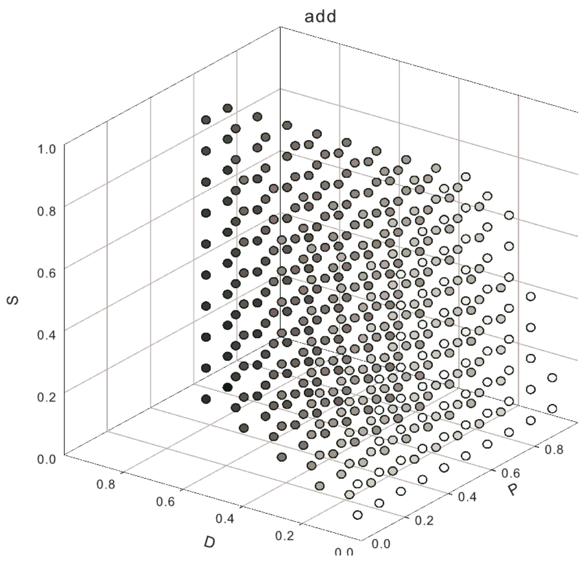
7.1.2.2 SystemC Simulator Construction The SystemC simulator construction is completed in two phases. In the first phase, the user specified C behaviors that correspond to the different RFID primitives are converted into SystemC designs. In the second phase, the compiler generates the simulator framework for the entire tag, which includes the unpacking, decode and packing logic. The simulator is made by compiling the SystemC design.

Generation of RFID Primitive Designs: The user specified C behaviors that correspond to the different RFID primitives are represented as SDFGs in the SuperCISC compilation flow, as explained in detail in Section 6.2. An SDFG is an extension of the more common control and data flow graph (CDFG) where basic blocks have been merged by converting control edges into data edges using hardware predication [34, 35].

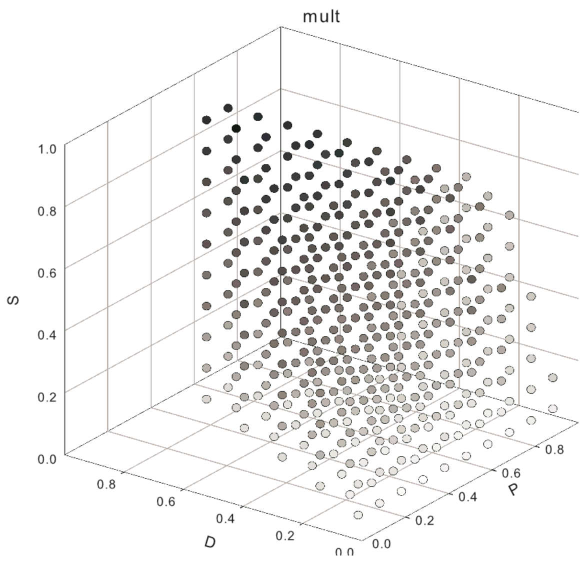
The original SuperCISC compilation flow was extended with a SystemC generation pass. The pass translates each super data flow graph (SDFG) into a behavioral SystemC design. This uses the SystemC abstract syntax tree (AST) of PACT HDL for the intermediate representation, which contains data structures that behave according to the SystemC specification. The highest AST data structure is a SystemC program. A program contains a top-level module or header and a cc file. A module contains its name, the headers, a global symbol table of signals, the module declaration items such as ports and a process block. A cc file is made of behavior declaration items, that describe the behavior of the items declared in the header file. Each item can be a process behavior, a method behavior or a global variable declaration. The process nodes contain local symbol tables with VHDL variables and statement nodes. The method nodes contain statement nodes, a list of arguments, etc. Statement nodes such as assignments, if statements, etc., are made of expressions and operators.

The SystemC AST was modified to compile within the SUIF2 compiler. This required changing the STL structures to use the SUIF STL replacement structures. The SUIF STL replacements are made to behave similarly to the STL structures but some of the methods from the STL structures are not defined.

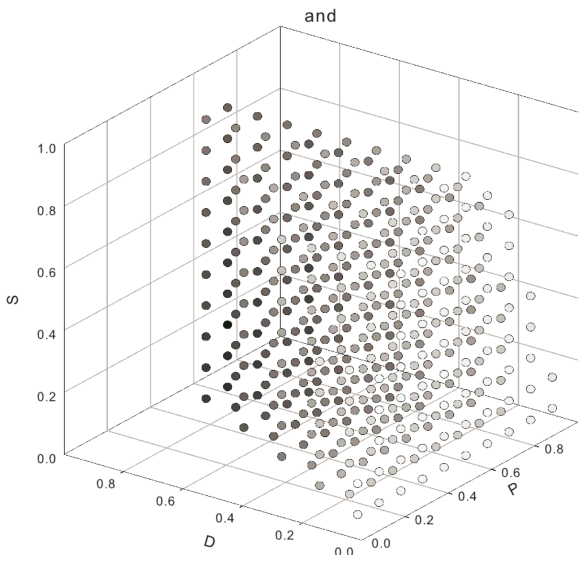
Figure 65 shows the outline for SystemC simulator generation from a control and data flow graph. A top-level SystemC module is created and ports are added for each input and output node. The module's SystemC code generation is expanded in Figure 66. In



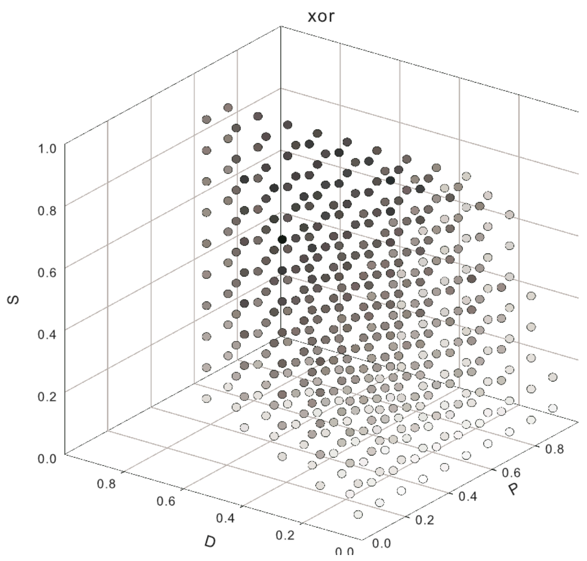
(a) Adder



(b) Multiplier



(c) AND Operation



(d) XOR Operation.

Figure 64: 4-D plots of p, d, and s versus power for functional units synthesized as 0.16μm OKI ASICs. Power is indicated as a color between black and white where solid black represents the least power consumed by the device and solid white indicates the most power consumed by the device. Measurements are taken at 0.1 intervals in each dimension p, d, and s.

```
sc_program *cdfg2sc(cdfg *graph) {
  // Create a new SystemC module, adding ports for each I/O node.
  sc_module *m = create module(graph);
  // Walk the control data flow graph and generate the module
  sc_program *p = walk cdfg(graph,m);
  // Generate .cc and .h files for the custom simulator.
  sc_print(p);
  // Generate the main program (testbench) to run the simulation
  sc_tb *tb = cdfg2sctb(graph);
  sc_print(tb);
}
```

Figure 65: Outline for SystemC simulator generation.

```
sc_program *walk cdfg(cdfg *graph, sc module *m) {
  // Get the basic block list
  list<bb *> *bblist = graph->get bblist();
  for (each bb in the bblist) {
    for (each node in the bb) {
      // Create unique module declaration generate SystemC
      if (have not seen this node type) {
        create module for this node type;
      }
      // Instantiate the module
      if (operation or mux) {
        instantiate the module;
      // Create a signal for every node within the cdfg
      sc_signal declr *sig = new sc_signal_declr(
                 convert type(node), convert name(node));
                 m->addDeclrItem(sig);
    }
    // Link bottom nodes to output ports
    connect outputs(graph, m);
  }
}
```

Figure 66: Outline for SystemC module generation.

this step, each node in the graph is examined and unique module declarations are added to the design. The modules' SystemC code is then generated and they are instantiated in the design. Finally, the bottom CDFG nodes are linked to the output ports.

Trace instructions are added to each node's design, which save information about the functional units associated with changing signal values. The generation of trace instructions is outlined in Figure 67. Once each node is constructed, the list of statements is passed to the *generateTraces()* function. This function examines each statement in the list such as assignment, if, switch, function call, etc, and invokes *traceInstrs()* with the expressions in the statement. The function *traceInstrs()* constructs and returns a list of trace statements for each expression recursively. All the trace instructions are appended to the original list of statements in the node.

Generation of The Simulator Framework: In the second phase, the RFID Compiler generates the simulator framework for the entire tag, which includes the unpacking, decode and packing logic. The compiler uses the information in the input macros file along with the SystemC AST data structures to generate this. An outline of the simulator framework generation is shown in Figure 67.

The SystemC hardware blocks generated in the first phase are instantiated in the design for the complete tag simulator. This is done by *addBehavComponent()*. The text file generated by the *printComponentDetails()* function, described previously in Section 6.2, is parsed, and the corresponding module declarations, instances and port mappings are constructed. Trace instructions are added to the functional units in the top-level design. The test-bench is then constructed, with instances of the top level module in the design and the signal generator. The signal generator drives the command line by reading from a command text file. The test-bench opens separate trace files for each of the primitive modules and finally runs the simulation.

7.1.2.3 Power Estimation

The power macromodeling flow automatically generates probabilistic input test vectors for use in the tag simulation. It uses the Markov chain-based sequence generator, used in Section 7.1.2.1, to generate test vectors with p, d, and s values that are evenly distributed in the 3-dimensional space. Since the actual commands

```
list<sc_statement *> generateTraces(statement list) {
  for(each statement in statement list) {
    if(stmt->isAssignment()) {
        list<sc_statement *> trlist = traceInstrs(expression);
        append trace instructions list to statement list;
    }
    if(stmt->isFunction()) {
        for(each argument in function) {
            list<sc_statement *> trlist = traceInstrs(argument);
            append trace instructions list to statement list;
        }
    }
    if(stmt->isIf()) {
        list<sc_statement *> trlist = traceInstrs(ifexpr);
        append trace instructions list to statement list;
        list<sc_statement *> then_traces = generateTraces(then statement list);
        list<sc_statement *> else_traces = generateTraces(else statement list);
        sc_if *i = new sc_if(ifexpr, then_traces, else_traces);
        append if statement to statement list;
    }
    if(stmt->isSwitch()) {
        list<sc_statement *> trlist = traceInstrs(switchexpr);
        append trace instructions list to statement list;
        for(each case) {
            list<sc_statement *> casetlist = generateTraces(case statement list);
            list<sc_statement *> trlist = traceInstrs(caseexpr);
            append trace instructions list to the case's trace list;
            sc_case *ci = new sc_case(caseexpr, casetlist);
            create switch and/or append case;
        }
        append switch statement to statement list;
    }
  }
  return statement list;
}
```

Figure 67: Outline for trace instruction generation.

```
void generateCode(opcodeList, decList) {
    sc_module *module = new sc_module(m_name);
    add input and output ports;
    sc_cc_file *ccfile = new sc_cc_file(m_name);
    sc_program *p = new sc_program(module, ccfile);
    create process block and declarations;
    add declr item to module;
    create process and add to block;
    create sensitivity list and add items;
    create processes csm1_clocked_proc, csm1_nextstate_proc, etc;
    // create behavior for taglogic
    // add signal declarations from symbol table
    for(each operand in symbol table) {
        sd = new sc_signal_declr(new sc_base_type(SC_LV, decList->pValue),
                                 new sc_name(tempsymb->operandName));
        module->addDeclrItem(sd);
        for(each nested operand) {
            sd = new sc_signal_declr(nested-type, nested-id);
            module->addDeclrItem(sd);
        }
    }
    // add behavior instances
    for(each opcode) {
        open opcode file;
        idec = new sc_instance_declr(new sc_name(opcodeName), inum);
        module->addDeclrItem(idec);
        addBehavComponent(fp, module, block, iname);
    }
    add statements;
    create decoder switch statement;
    for(each opcode) {
        create case;
        add unpacking statements for each operand in command;
        add packing statements for each operand in response;
    }
    construct behavior for all processes;
    add trace instructions;
    // generate test bench
    sc_tb_file *tb = generate_tb(opcodeList);
    sc_print_h(p);
    sc_print_cc(p);
    tb->print(tbfile);
}
```

Figure 68: Outline for simulator framework generation.

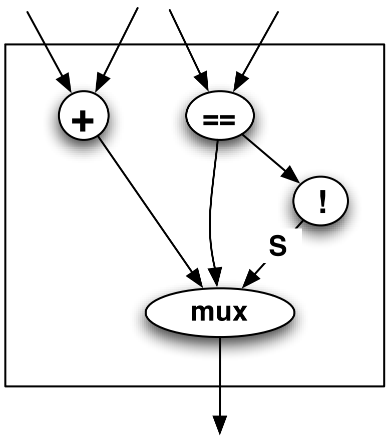


Figure 69: An example module. Energy of the module is based on the energy of each of the individual functional units addition, equivalence, multiplexer, and logical not.

issued by the RFID reader may be unknown at the time of simulation, sequences with all possible statistics are applied to the tag simulation. However, while designing tags for a given RFID system, the user may be able to pre-determine the expected workloads generated by the RFID reader. The user can use these input vectors instead of the probabilistic vectors for designing tags that are power-optimized for the actual workloads.

When the simulation is executed, a trace file is generated with the program execution statistics. If the signal values at a functional unit change, a trace instruction records the unit's identification, its signal values and the time the unit spent in the calculation. During power estimation, each trace instruction is read and the module power is constructed behaviorally. The power estimation is illustrated using a simple example module, shown in Figure 69.

The energy calculation for the module is based on the energy consumed by each of the functional units. In the example, this consists of an adder, an equivalence checker, a selector, and logical not. Consider the adder unit. Based on its inputs, the *p*, *d*, and *s* values are computed using the technique described in [46]. For the inputs (0,1), the computed *p*, *d*, and *s* values are 0.017, 0.031, and 0.061. The power consumed by the adder for these values is obtained by looking up the library, and is $1.14 \cdot 10^{-6}$ W. The corresponding energy is calculated as the power multiplied by the time spent in executing the add operation. Similarly, the power consumed by the equivalence operator, logical not operator, and the multiplexer units are $1.09 \cdot 10^{-6}$ W, $1.22 \cdot 10^{-6}$ W, and $1.14 \cdot 10^{-6}$ W, respectively. The energy of the module is constructed by aggregating the individual energies. Energy is averaged over the simulation time to calculate power.

7.1.2.4 Results We compared our power macromodeling approach to generating a design and power profiling it with existing tools. The flow we compared to was design synthesis using Synopsys Design Compiler and power profiling by simulating the design in Mentor Graphics ModelSim to generate switching information and power estimation using Synopsys PrimePower.

Table 15 shows the run times and power estimated with the power macromodeling technique and the traditional power estimation method for the Part 6C inventory primitives. As can be seen from the table, the power consumption is estimated to be very similar between both techniques. The power estimated by the power macromodeling technique is slightly higher, and is 4.76% on average. However, the calculation time was improved by 108, on average.

Table 16 shows the run times and power estimated using the above methods for some of the ISO 18000-7 primitives. These primitives consume much less power than similar Part 6C primitives. For instance, compare the *Collection* command from ISO 18000 Part 7 and the *Query* command from ISO 18000 Part 6C standards. These commands have similar functionality but use a significantly different protocol. Both the power estimation methods show the significant advantage of *Collection* over *Query* for power. However, the run time was improved by 241 times and 89 times, respectively. For the ISO 18000-7 commands, the power estimated by the power macromodeling technique is, on an average, 6.84% more than that estimated by the traditional method. The calculation time was improved by 251, on average. For full tag designs, this technique provides an answer in seconds while the other technique can take minutes or hours.

Table 15: Power Macromodeling vs. Traditional Method For Commands From ISO 18000-6C.

Primitives	**Query**	**QueryAdj**	**QueryRep**	**Nak**	**Ack**
Power Macromodeling					
Power (mW)	0.120	0.061	0.063	0.036	0.057
Time (s)	0.46	0.34	0.33	0.27	0.35
Traditional Method					
Power (mW)	0.113	0.059	0.061	0.034	0.052
Time (s)	40.71	36.02	36.92	34.75	36.11
Times Speedup	89	106	112	129	103
Power Difference	6.19%	3.04%	3.61%	4.40%	10.49%

Table 16: Power Macromodeling vs. Traditional Method For Commands From ISO 18000-7.

Primitives	**Collection**	**Sleep**	**OwnerId**	**UserId**	**Status**
Power Macromodeling					
Power (uW)	0.0114	0.0086	0.0086	0.0057	0.0084
Time (s)	0.13	0.12	0.12	0.13	0.11
Traditional Method					
Power (uW)	0.0113	0.0079	0.0079	0.0051	0.0080
Time (s)	30.31	29.45	29.24	32.38	30.66
Times Speedup	241	243	246	257	269
Power Difference	0.53%	8.22%	8.22%	12.40%	4.86%

Table 17: The impact of modifying a standard can be studied using the Power Macromodeling Flow. Results shown are for reducing the number of states in the Part 6C standard state machine to six. On average, power is reduced by 10.97%. Similarly, many different modifications can be modeled.

Primitives	**Query**	**QueryAdjust**	**QueryRep**	**Nak**	**Ack**
Original					
Power (mW)	0.120	0.061	0.063	0.036	0.057
Time (s)	0.46	0.34	0.33	0.27	0.35
Modified					
Power (mW)	0.115	0.054	0.055	0.030	0.051
Time (s)	0.46	0.33	0.31	0.25	0.34
Power Difference	-4.17%	-11.48%	-12.84%	-16.01%	-10.37%

7.1.3 Evaluation of Alternate Protocol Designs: Example

Many design parameters in the protocol such as reducing or adding a state, adding or removing a variable, size of command code, size of variables, etc may have an impact in the power consumed by the final implementation. The impact may be significant depending on the behavior of each command with respect to the state or variable. The power macromodeling flow can be used to evaluate such variations in protocol designs.

For example, consider the Part 6C tag. The tag state machine has seven states as explained in Section 6.4. Almost every primitive behavior implements different actions in each state. Table 17 shows results for reducing the number of states in the Part 6C standard state machine to six. It can be seen that all the primitives consumed lesser power as a result of the modification. The difference is highest for the *Nak* command and lowest for the *Query* command. The average decrease in power is 10.97%, which is significant. Thus power analysis of alternate protocols can be done effectively and in a very short time.

7.2 STUDY OF APPLICATION SPECIFIC INSTRUCTION SETS FOR MICROPROCESSORS

Application Specific Instruction Processors can achieve much better levels of performance and power efficiency than general purpose processors, since they contain only those capabilities necessary to execute certain target workloads [47, 48].

For the microprocessor-based implementation of RFID primitives, power may be reduced by designing an ASIP. Reducing the size of the instruction set architecture of the target embedded processor can significantly reduce the amount of logic for instruction decode as well as settings to the remainder of the processor. This generally translates into reduced power consumption. In some cases removal of high-power consuming capabilities (e.g. multiplication) can also result in significant power savings. In this dissertation, the instruction set architecture of the EISC and ARM processors are studied and the number of instructions used by the RFID program are evaluated.

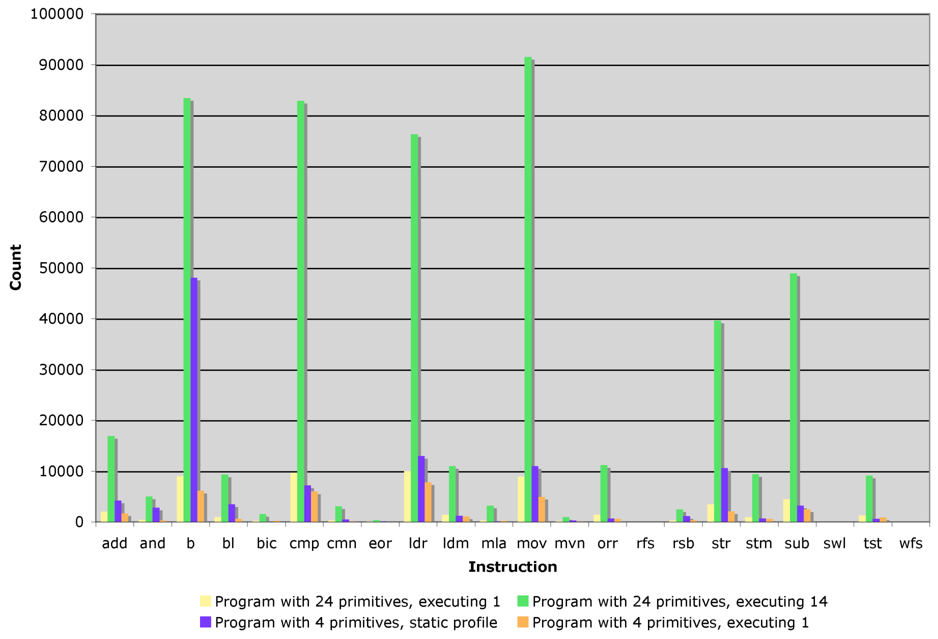


Figure 70: ARM instruction profile.

7.2.1 Power and Area study of ISA subset of EISC CPU

SimpleScalar's sim-profile tool was used to obtain ARM instruction and instruction class profiles for RFID tag controller software. Figure 70 shows a profile of ARM instructions for different runs of the RFID program. Since an instruction set simulator for EISC was unavailable, the instructions were only statically profiled. Figure 71 shows a profile of EISC instructions for the 4 primitive RFID program.

The total number of instructions in the EISC ISA is 56[49]. The number of instructions used by the static instruction profile of the RFID program is 38. If the remaining instructions can be removed from the ISA, the instruction decode, execute and data path will become simpler. Since the architecture of EISC is not known, we assume that there is a linear reduction of an arbitrary percentage of the total power per instruction. Based on this, we estimate the impact of removing instructions. Assume N% of power is consumed by Decode + Execute + Data Path. From Table 18, I are the subset of EISC instructions that are

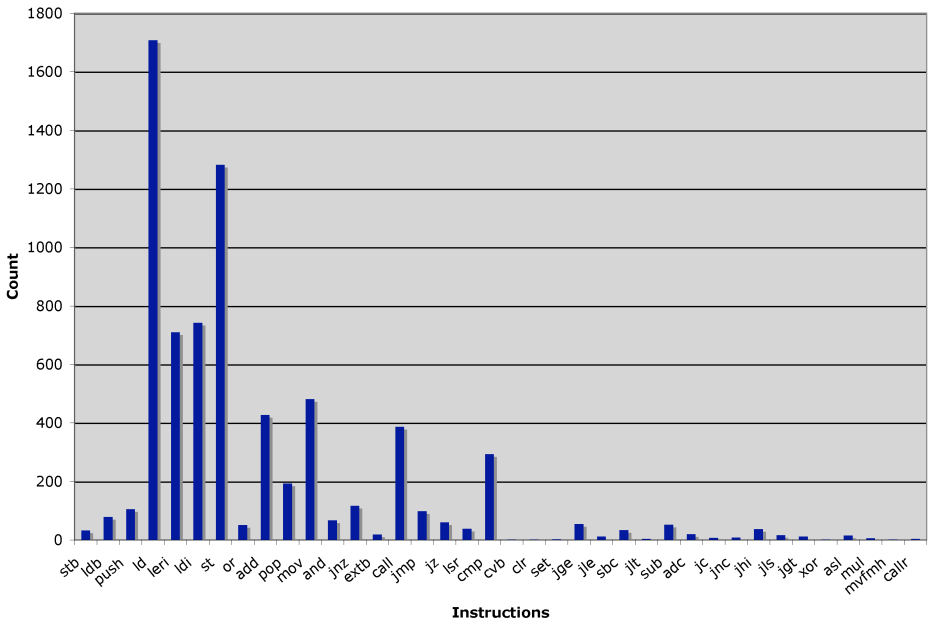


Figure 71: EISC instruction profile.

Table 18: Power study for reducing EISC ISA.

Total number of instructions in ESIC ISA, I	56
Instructions used by RFID program, I'	38
I'/I	0.68

% of Total Power for Decode + Execute + Data Path, N	**5**	**10**	**15**	**Original ISA**
Power of N (mW)	1.48	2.95	4.43	N/A
Reduction in Power of N with Reduced ISA (mW)	0.47	0.95	1.42	N/A
After optimization:				
Energy consumption for 4 Primitive Program (uJ)	17.01	16.73	16.45	17.29
Energy consumption for 12 Primitive Program (uJ)	19.16	18.84	18.53	19.47

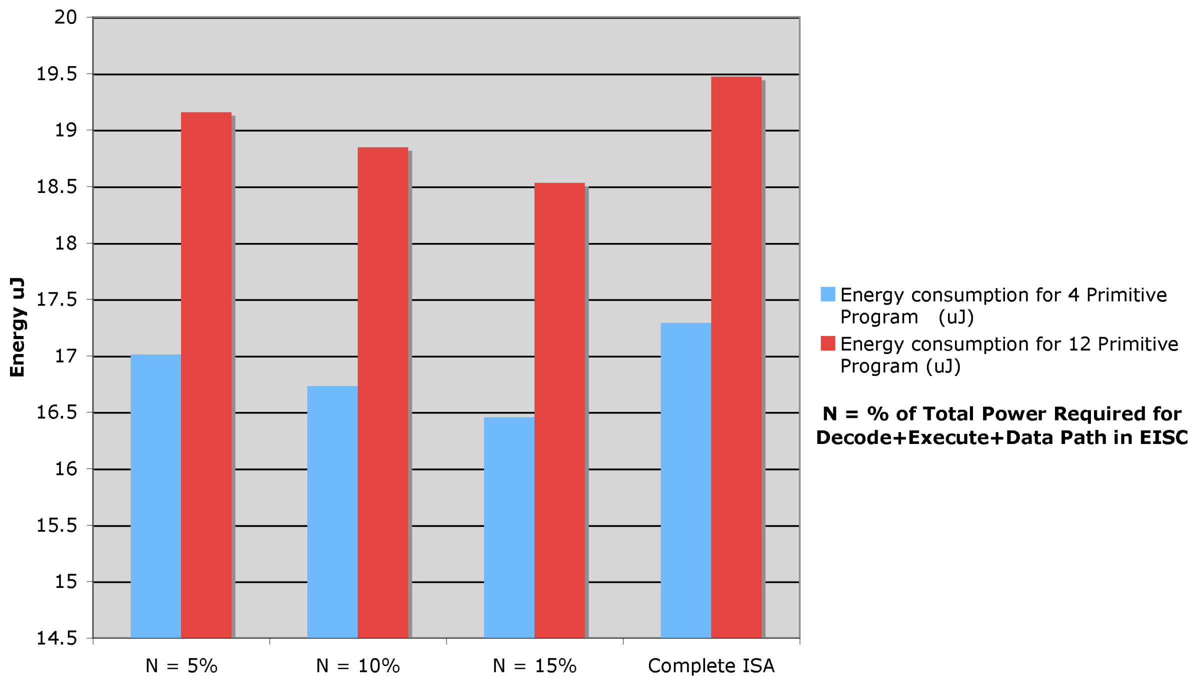


Figure 72: Energy consumption of RFID C program generated by the compiler for different values of N.

required by the system. Power of N is calculated as N% of 29.5mW. Then the reduction in power of N with reduced ISA is (Power of N Power of N * 0.68). Using the new reduced power the energy consumption is calculated as Power * Run Time of RFID program.

Different values of N have been chosen to indicate the different fractions of power that could be consumed by Decode + Execute + Data Path in EISC. The energy consumption of the two-benchmark programs have been calculated based on the different values of N. Table 18 shows a summary of the above calculations. Figure 72 shows the energy consumption of the RFID program for different values of N. It can be seen that the power consumption, and therefore the energy, of the RFID program on EISC can be reduced by removing instructions that are not required from the ISA.

We have presented a study for potential power reductions by reducing the size of the instruction set architecture of EISC. It is possible to further explore power optimizations for the microprocessor-based tag. It is also possible to design the processor to be sensitive to commonly occurring patterns in the resulting software run on the processor. The instruction encoding can be adjusted to reduce the number of bits required to encode an instruction, as well as to reduce the bit changes between different instructions which impact power consumption.

8.0 CONCLUSIONS

Radio Frequency Identification (RFID) tags are being used in an increasingly wide variety of applications. Although standards exist, custom tags, with additional capabilities, are sometimes required for specific applications. However, the lack of supportive design environments for these custom systems results in an unacceptably long deployment time for leveraging the benefits of RFID technology. To significantly reduce the design time, it is necessary to develop the design automation tools that allow the designers to capture a specification more quickly as well as reducing the time necessary to move from specification to hardware implementation.

This objective has been achieved in this dissertation by the RFID specification methodology and compilation flow, which permits automated design of low-power RFID tags. The main contributions of this dissertation include:

- developing and implementing high-level design entry using a simple description of RFID primitives and their behavior in ANSI-C, and a back-end capable of targeting microprocessor-based tags,
- developing and implementing a front-end capable of supporting behavior specification in VHDL and a back-end capable of targeting custom hardware-based tags,
- developing and implementing a back-end integrated with the SuperCISC compilation flow, and thus allowing the RFID compiler to automatically convert the user supplied behavior in C to low power synthesizable VHDL optimized for RFID applications, and extending it with features to support primitives from the ISO 18000-6C protocol, and,
- developing and implementing a power macromodeling flow, which calculates power at a high level during the RFID compiler design automation process.

8.1 CONTRIBUTIONS

8.1.1 RFID Compiler for the Microprocessor-based Tag

For high-level design entry, simple assembly-like description of the standards, or *RFID macros* were designed. A parser was built to read this and build it in to the compiler. To enter the tag behavior in response to each *RFID primitive*, ANSI-C was chosen. To simplify the user interaction, the parser automatically generates C code templates, that indicate where the user must specify such custom behavior. The user uses simple ANSI-C constructs to plug in the behavior into the template.

The first compiler back-end was developed and implemented to target microprocessor-based tags. The RFID Compiler generates the tag controller code based on the input *RFID macros* and the tag behavior in the C language. The output C code is compiled using an embedded compiler to generate executable code for the microprocessor target integrated with the tag. Three low-power embedded microprocessors were selected and power profiled in Chapter 4. In terms of energy consumption, StrongARM is consistently 4-6 times worse than XScale requiring hundreds of μJ compared to tens of μJ for XScale. For ease of comparison, consider program B which contained 12 primitives. This program was run or modeled successfully on all platforms. Of the microprocessors, StrongARM performed the worst, requiring about 4 times more energy than an XScale and almost 10 times that of an EISC. However, for the low-power EISC processor, system memory is a limitation.

8.1.2 Hardware RFID Compiler with VHDL Behavior

The overhead of using a microprocessor-based controller for the RFID tag is considerable compared to a hardware-based controller. The ARM-based processors require hundreds of instructions to be executed to generate the response for a single primitive, resulting in significant energy usage compared to custom hardware-based designs. While the EISC processor is a low-power processor option, system memory is a limitation. To improve both the power and capacity of the controller, the hardware-based RFID compiler was described in Chapter 5. The hardware RFID compiler processes the *RFID macros* to generate VHDL template files and the final tag controller in VHDL. Xilinx Spartan 3, Coolrunner II and Actel Fusion devices were targeted for tag designs. The energy required by the Coolrunner is in the nJ range while the best performing processor (EISC) still requires 17 μJ, for the same primitives. The ASIC implementation drops the energy consumed down to 0.07 nJ per transaction when active, which is comparable to the energy used by commercial active tag ASICs. The area of the ASIC controller is less than 4300 cells for a 40 primitive controller, which is also comparable to the conventionally designed active tags used in the industry.

8.1.3 Hardware RFID Compiler with C Behavior

Because C is a significantly more universally known language than VHDL or Verilog, it is desirable to have the user specify the primitive behaviors for the RFID Tag in C while generating a custom hardware in VHDL. In order to synthesize C-based primitives into VHDL, the back-end of the hardware RFID compiler is integrated with the SuperCISC compilation flow in Chapter 6. The SuperCISC compiler and the RFID compiler were extended so that the behavioral VHDL components can be instantiated and combined with the automatically generated tag hardware. The ASIC designs implemented using the two hardware RFID Compilers are compared in terms of their areas and power consumption. There is a nominal increase in area when C behavior is used instead of direct VHDL, and requires 2.91% and 9.37% more area for the tags with a single primitive and twenty four primitives of ISO 18000-7, respectively. This area overhead can be reduced through the use of specialized controls for resource sharing in the design compiler tool.

Different standards such as the ISO 18000-7, ISO 18000-6C, ANSI, ISO 18185, etc were implemented using the RFID compiler. The communication primitives of ISO 18000-6C (Gen-2) standard are significantly different and more complex than those of the ISO 18000-7 standard. The standard relies on intermediate storage and storage of state at several points during each communication operation. The states and keys of the target device are used to facilitate operations such as tag singulation, collision arbitration, and security encoding. We compare the ASIC implementations of the ISO 18000 Part 6C *Query* command and the ISO 18000 Part 7 *Collection* command, which have similar functionality. The *Query* implementation is not only much larger and consumes significantly more power than the *Collection* command implementation, but also requires more power than ten combined ISO 18000 Part 7 commands.

8.1.3.1 EPC C1 Generation 2 Hardware RFID Compiler The C behavior-based hardware RFID compiler was extended to support the Part 6C standard in Chapter 6.4. The compiler-generated Part 6C inventory and access primitives were targeted to the Xilinx Spartan 3 FPGA and ASICs and compared with corresponding manual designs. For the tag Spartan 3 design, the resource utilization is almost the same for both the approaches and improves slightly with the RFID compiler-based approach. For the tag ASIC design, the area was found to increase by up to 66.56%, due to the large number of variables and the complex behaviors. This is in part due to how design compiler does resource sharing. To reduce this overhead, area optimizations have been built in to the RFID Compiler. A precision pass updates the sizes of the CDFG input and output nodes, followed by a conversion pass that examines each operation node and reduces its size to the minimum bit width calculated based on its inputs. With these optimizations the area increase is substantially reduced to 10.55%. Thus, the use of the design automation tool can make the design of the complex Part 6C tags faster, while achieving tag implementations that are very similar in area and power to the corresponding manual designs.

8.1.4 Techniques For Optimizing The Tag Power Consumption

Power optimization is critical in RFID systems because the power budget is limited. However, during the design of RFID tags, power consumption has been held as the secondary metric, next to cost. Further more RFID protocols are typically designed without taking into account many of the area and power consumption impacts of their final implementation. This is partly because it takes months or years of engineering effort to gain an accurate estimate of power consumed for a tag or a protocol implementation, with the existing design flows.

This dissertation enables the programmer to tackle the power problem early in the design flow by describing a power macro-modeling technique that works in concert with the RFID design automation flow. The power macro-modeling flow, described in Chapter 7:

- calculates power at a high level during the RFID compiler design automation process
- generates estimates which are within 15% accuracy and about one hundred times faster than the conventional ASIC power estimation flows
- allows optimization of the primitives and / or the behaviors and the effective evaluation of alternate protocol designs
- does not require detailed knowledge of ASIC synthesis and power estimation flows

The power macromodeling component integrates into the RFID compiler, which reads the RFID protocol description, including the RFID macros and their corresponding behavior in C and automatically generates a hardware design for the tag in the SystemC language. The SystemC description of the hardware design is compiled into a custom simulator executable for that particular hardware design. SystemC was chosen because the various steps in power macromodeling are easily and seamlessly integrated into the simulator using C++ code. The final simulator is typically fast as the simulator is a compiled binary rather than an interpreted simulation of hardware description languages.

The SystemC simulator construction is completed in two phases. In the first phase, the user specified C behaviors that correspond to the different RFID primitives are converted into SystemC modules. For this, a pass that converts the control and data flow graphs into SystemC code was implemented in the SuperCISC compiler. In the second phase, the RFID compiler generates the simulator framework for the entire tag. For this, the RFID compiler was extended with a back-end to generate the SystemC simulator and the SystemC modules are integrated into the framework. After the SystemC simulator is generated, it is compiled into a binary using a software compiler. Probabilistic input test vectors are then automatically generated for use in simulation. The compilation flow executes the simulation and generates annotated trace files with information about all the functional units in the design. These functional units have been pre-profiled for power based on various input parameters. The power estimation of each RFID primitive and the overall tag design is calculated by combining the trace information with the profiles to determine each units power. The resultant power estimates correspond to the actual activity of the tag behavior in hardware.

The power macromodeling flow has been used to implement and profile primitives from the ISO 18000-7 and 18000-6C standards. These were compared to VHDL primitives synthesized using the conventional methods. For both the standards, the power consumption is estimated to be very similar between both techniques. For the ISO 18000-6C commands, the power estimated by the power macromodeling technique is slightly higher, and is 4.76% on average. However, the calculation time was improved by a factor of 108, on average. For the ISO 18000-7 commands, the power estimated by the power macromodeling technique is, on an average, 6.84% more than that estimated by the traditional method. The calculation time was improved by a factor of 251, on average. For full tag designs, this technique provides an answer in seconds while the conventional technique takes minutes or hours.

Many design parameters in the protocol such as reducing or adding a state, adding or removing a variable, size of command code, size of variables, etc may have an impact in the power consumed by the final implementation. The impact may be significant depending on the behavior of each command with respect to the state or variable. As an example, the power macromodeling flow was used to evaluate the result of reducing the number of states in the Part 6C standard state machine to six. All the primitives consumed less power as a result of the modification, on average 10.97% and the average run time was only 0.34 s. Thus, using the power macromodeling flow, power analysis of alternate protocols can be done effectively and in a very short time.

Thus, this dissertation created an easier and faster way for RFID application programmers to develop low-power RFID tags for their applications.

8.2 FUTURE RESEARCH

The RFID compiler developed in this dissertation automates the design of the tag controller, while the physical layer encodings are implemented manually. The physical layer protocol also varies across different standards. One possibility of extending this dissertation would be to integrate the compiler with the automatic generation of physical layer decoder and encoder blocks specified in [3].

In this dissertation, the RFID compiler was used to implement tag controllers with communication primitives from different standards such as the ISO 18000-7, ISO 18000-6C, ANSI and ISO 18185. The RFID compiler also has the capability of implementing controllers for RFID readers. The communication primitives used by the readers can be similarly input to the compiler to automatically generate the corresponding controllers. The compiler is also applicable to other wireless devices such as digital cell phones, blue tooth devices, etc., in which the communication can be decomposed into primitives.

The existence of multiple keys and intermediate states and the complexity of primitives in the ISO 18000 Part 6C protocol presents challenging design problems compared to the simpler ISO 18000 Part 7 protocol. The use of the RFID compiler can make the design of these tags faster, however, the original complexity is not reduced. One solution is to examine the needs of the protocol and explore modifications to the protocol that leads to simpler implementations. A second possibility is to examine ways to decompose the CISC-like primitives in the standard to create a more RISC-like implementation.

While the power macromodeling technique is targeted towards hardware-based tags, an application specific instruction processor (ASIP) is studied for microprocessor-based tags, and shows a possible reduction of power in their design. One possibility of extending this dissertation is to provide a tool for implementing an ASIP tag tailored for the application. The power macromodeling flow has been used to evaluate variations in protocol designs. Another possibility of extending this dissertation is to use the flow to study the existing standards and to propose modifications to them or even to design a new standard.

BIBLIOGRAPHY

[1] Savi Technology, “Savi Technology,” website, http://www.savi.com.

[2] T. Yoshihisa, Y. Kishino, T. Terada, M. Tsukamoto, R. Sagara, T. Sukenari, D. Taguchi, and S. Nishio, “A rule-based rfid tag system using ubiquitous chips,” in *Proc. The 2005 International Conference on Active Media Technology, AMT 2005*, May 2005, pp. 423 – 428.

[3] S. Dontharaju, S. Tung, R. R. Hoare, J. T. Cain, , M. H. Mickle, and A. K. Jones, *Design Automation for RFID Tags and Systems*, S. Ahson and M. Ilyas, Eds. Taylor and Francis.

[4] *EPC Radio-Frequency Identity Protocols: Class-1 Generation-2 UHF RFID Protocol for Communications at 860 MHz - 960 MHz*, Version 1.0.9 ed., EPCglobal, Inc., January 2005.

[5] S. Tung, S. Dontharaju, L. Mats, P. J. Hawrylak, J. T. Cain, M. H. Mickle, and A. K. Jones, *Layers of Security for Acitve RFID Tags*, S. Ahson and M. Ilyas, Eds. Taylor and Francis.

[6] Texas Instruments, “Texas instruments’ RFID technology streamlines management of vatican library’s treasured collections,” 2004, www.ti.com/tiris/docs/news/news_releases/2004/rel07-07-04.shtml.

[7] L. M. Ni, Y. Liu, Y. C. Lau, and A. P. Patil, “LANDMARC: indoor location sensing using active RFID,” *Wireless Networks*, vol. 10, November 2004.

[8] C. Li, L. Liu, S. Chen, C. C. Wu, C. Huang, and X. Chen, “Mobile healthcare service system using RFID,” in *IEEE International Conference on Networking, Sensing and Control*, vol. 2, 2004.

[9] A. Cerino and W. P. Walsh, “Research and application of radio frequency identification (RFID) technology to enhance aviation security,” in *Proc.s IEEE National Aerospace and Electronics Conference*, 2000.

[10] International Standards Organization, “ISO/IEC FDIS 18000-7:2004(e),” Standard Specification, 2004.

[11] American National Standards Institute, "ANSI NCITS 236:2001," Standard Specification, 2002.

[12] International Standards Organization, "ISO/IEC FDIS 18185-1:2006," Standard Specification, 2006.

[13] ——, "ISO/IEC FDIS 18000-6:2004/amd 1:2006(e)," Standard Specification, 2006.

[14] Savi Technology, "SaviTag ST-602," Datasheet, 2004.

[15] S.-M. Wu, J.-R. Yang, and T.-Y. Liu, "An asic for transponder for radio frequency identification system," in *Proc. Ninth Annual IEEE International ASIC Conference and Exhibit, 1996*, Sept. 1996, pp. 111–114.

[16] U. Kaiser and W. Steinhagen, "A low power transponder ic for high performance identification systems," *IEEE Journal of Solid-State Circuits*, vol. 30, no. 3, pp. 306–310, March 1995.

[17] R. Howes, A. WilIiams, and M. Evan, "A read/write rfld tag for low cost applications," in *Proc. IEE Colloquium on RFID Technology*, 1999, pp. 4/1–4/4.

[18] R. Imura, "The world's smallest rfid u-chip, bringing about new business and lifestyles," in *Proc. 2004 Symposium on VLSI Circuits*, June 2004, pp. 120–123.

[19] M. Usami, "An ultra small rfid chip: u-chip," in *Proc. 2004 Symposium on Radio Frequency Integrated Circuits (RFIC)*, June 2004, pp. 241 – 244.

[20] K. Takaragi, M. Usami, R. Imura, R. Itsuki, and T. Satoh, "An ultra small individual recognition security chip," *IEEE Micro*, vol. 21, no. 6, pp. 43–49, Dec. 2001.

[21] M. Kohvakka, M. Hannikainen, and T. D. Hamalainen, "Wireless sensor prototype platform," in *Proc. The 29th Annual Conference of the IEEE Industrial Electronics Society, IECON '03*, Nov. 2003, pp. 1499 – 1504.

[22] IEEE Computer Society, "IEEE Standard 802.3-2005," December 2005.

[23] American National Standards Institute, "ANSI/TIA/EIA-422-B," Standard Specification, May 1994.

[24] ATIS Committee T1A1, "ATIS Telecom Glossary 2000," Alliance for Telecommunications Industry Solutions (ATIS), Tech. Rep. T1.523-2001, 2001.

[25] A. K. Jones, R. Hoare, S. Dontharaju, S. Tung, R. Sprang, J. Fazekas, J. T. Cain, and M. H. Mickle, "An automated, FPGA-based reconfigurable, low-power RFID tag," *Journal of Microprocessors and Microsystems*, vol. 31, no. 2, pp. 116–134, March 2007.

[26] Intel, "SA-110 microprocessor technical reference manual," 1998, ftp://download.intel.com.

[27] ——, "Intel PXA27x processor family developers manual," 2004, ftp://download.intel.com.

[28] Y. Cha, "EISC core," Presentation to University of Pittsburgh, February 2005.

[29] Sim-panalyzer, "Simplescalar-ARM power modeling project," http://www.eecs.umich.edu/ panalyzer.

[30] C. Gilberto, M. Martonosi, J. Peng, R. Ju, and G. Lueh, "XTREM: A power simulator for the intel xscale core," in *Proc. 2004 ACM SIGPLAN/SIGBED conference on Languages, Compilers and Tools*, vol. 39, no. 7, June 2004, pp. 115–125.

[31] J. Russell and M. Jacome, "Software power estimation and optimization for high performance, 32-bit embedded processors." [Online]. Available: citeseer.ist.psu.edu/55191.html

[32] A. Jones, D. Bagchi, S. Pal, X. Tang, A. Choudhary, and P. Banerjee, "Pact hdl: a c compiler targeting asics and fpgas with power and performance optimizations," in *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems.* ACM Press, 2002, pp. 188–197.

[33] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *Proc. International Symposium on Field Programmable Gate Arrays (FPGA)*, 2006, pp. 21–30.

[34] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster, "An FPGA-based VLIW processor with custom hardware execution," in *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2005, pp. 107–117.

[35] R. Hoare, A. K. Jones, D. Kusic, J. Fazekas, J. Foster, S. Tung, and M. McCloud, "Rapid VLIW processor customization for signal processing applications using combinational hardware functions," *EURASIP Journal on Applied Signal Processing*, vol. 2006, pp. Article ID 46 472, 23 pages, 2006.

[36] X. Tang, T. Jiang, A. Jones, and P. Banerjee, "Compiler optimizations in the pact hdl behavioral synthesis tool for asics and fpgas," in *IEEE International SoC Conference (IEEE-SOC)*, September 2003.

[37] S. Gupta, N. D. Dutt, R.K.Gupta, and A.Nicolau, "Spark : A high-level synthesis framework for applying parallelizing compiler transformations," in *International Conference on VLSI Design*, 2003.

[38] A. K. Jones, R. Hoare, D. Kusic, G. Mehta, J. Fazekas, and J. Foster, "Reducing power while increasing performance with supercisc," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 5, no. 3, pp. 1–29, August 2006.

[39] Xilinx Inc., "Linear feedback shift register v3.0, LogiCore."

[40] S. Gupta and F. N. Najm, “Power macromodeling for high level power estimation,” in *DAC ’97: Proceedings of the 34th annual conference on Design automation.* ACM Press, 1997, pp. 365–370.

[41] A. K. Jones, X. Tang, and P. Banerjee, “Compile-time simulation for low-power optimization using systemc,” in *IASTED International Conference on Modeling and Simulation*, 2004.

[42] X. Tang, T. Jiang, A. Jones, and P. Banerjee, “High-level synthesis for low power hardware implementation of unscheduled data-dominated circuits,” *Journal of Low Power Electronics*, vol. 1, no. 3, pp. 259–272, December 2005.

[43] “Overview of the systemc initiative,” SystemC Website, `www.systemc.org`.

[44] Z. Chen and K. Roy, “A power macromodeling technique based on power sensitivity,” in *DAC ’98: Proceedings of the 35th annual conference on Design automation.* ACM Press, 1998, pp. 678–683.

[45] X. Liu and M. C. Papaefthymiou, “A markov chain sequence generator for power macromodeling,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, July 2004.

[46] ——, “A static power estimation methodology for ip-based design,” in *Design, Automation, and Test in Europe*, March 2001, pp. 280–287.

[47] C. W. L. Wu and T. Austin, “Cryptomaniac: A fast flexible architecture for secure communication,” in *International Symposium Computer Architecture*, June 2001, pp. 110–119.

[48] H. Z. N. Clark and S. Mahlke, “Processor acceleration through automated instruction set customization,” in *International Symposium on Microarchitecture (MICRO)*, December 2003.

[49] A D Chips., “The se1608a hardware,” http://www.adc.co.kr/.