FPGA implementation of a wireless sensor node.

Junsong Liao

University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

https://scholar.uwindsor.ca/etd/7110

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000 ext. 3208.
FPGA Implementation of A Wireless Sensor Node

by

Junsong Liao

A Thesis
Submitted to the Faculty of Graduate Studies through Electrical and Computer Engineering in Partial Fulfillment of the Requirements for the Degree of Master of Applied Science at the University of Windsor

Windsor, Ontario, Canada
2007
NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.
Abstract

Real world wireless sensor network applications impose a wide range of constraints on the system platforms, such as size, cost, power consumption, performance and flexibility. Therefore, sensor network platforms tend to be application specific rather than general purpose. However, most academic research today still uses general CPU-based embedded systems as sensor network platforms. Although CPU-based sensor nodes are flexible and convenient for developing sensor network applications, they are inefficient in handling event-driven tasks and consequently consume more energy. Custom hardware, on the other hand, is more efficient in handling event-driven tasks, but less flexible than CPU-based platforms. In this thesis, a hardware-based sensor network node is presented as an alternative sensor network application platform capable of leveraging the event-driven nature of sensor network applications. The implementation is based on Zigbee standard and is targeted for the Xilinx Spartan3 FPGA. The FPGA implementation provides the flexibility of exploring design space while keeping the performance of application specific hardware. Significant improvements are achieved by avoiding the overhead of operating system and interrupt handling. Compared to traditional CPU-based sensor nodes, the hardware-based system has 3~11 times reduction in cycle count to execute frequently used sensor network tasks.
To my family for their unending support.
I would like to express my sincere gratitude for the invaluable guidance and constant support of my supervisors, Dr. Mohammed A.S. Khalid and Dr. Kemal E. Tepe. I would also like to express my gratitude to other members of my committee Dr. Ziad Kobti and Dr. Jonathan Wu for their kindness, flexibility and helpful comments.

I would like to thank my fellow graduate students of the ECE department for their support and advice throughout the course of my study and research. Especially, I wish to thank Hongmei, Omar, Marwan, Aws and Thuan for creating a friendly, encouraging research environment in the FPGA research lab. Working with them was a great experience.
# Contents

Abstract iv  
Dedication v  
Acknowledgments vi  
List of Figures x  
List of Tables xii  
List of Abbreviations xiii  

1 Introduction 1  
1.1 Wireless Sensor Networks .............................................. 1  
1.2 Thesis Objective ............................................................ 5  
1.3 Thesis Organization .......................................................... 6  

2 Background and Previous Work 7  
2.1 Dedicated Embedded Sensor System ................................. 8  
2.2 Dedicated Processor for Sensor Network Application ........... 11  
2.3 Dedicated Architecture for Sensor Network Application ........ 12  
2.4 Motivation of Our Work .................................................... 14
CONTENTS

3 System Architecture ................................................. 16
  3.1 Event-Driven Architecture ........................................... 17
  3.2 Hardware Acceleration .............................................. 21
  3.3 Parallel Distributed Computation ................................. 21
  3.4 System Components .................................................. 22
    3.4.1 Radio control unit ........................................... 23
    3.4.2 Parsing and classification unit .............................. 24
    3.4.3 Network control unit ........................................ 26
    3.4.4 Application and management unit .......................... 28
    3.4.5 Modification and framing unit ............................... 29
    3.4.6 CC2420 module ............................................... 29

4 Implementation and Results .................................... 30
  4.1 Implementation Platform ........................................... 30
    4.1.1 RC10 development board .................................. 30
    4.1.2 CC2400 evaluation board and CC2420 ....................... 33
  4.2 Implementation Details ............................................. 35
  4.3 Test Results .......................................................... 35
    4.3.1 Functional tests .............................................. 36
    4.3.2 Packet error rate versus communication range ............... 38
    4.3.3 Crosstalk induced packet error ............................... 38
    4.3.4 Performance analysis using execution cycle count ............ 41
    4.3.5 System throughput ............................................ 43
    4.3.6 The trade-off of hardware-based system ..................... 43

5 Conclusion and Future Work ................................. 45

References ......................................................... 47

Appendix ............................................................. 50

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
## List of Figures

1.1 Zigbee protocol architecture ................................................................. 4

2.1 Dedicated embedded sensor node system architecture .................. 9
2.2 Dedicated Embedded Sensor Node System Architecture .............. 10
2.3 SNAP architecture .................................................................................. 11
2.4 Event-processor-based architecture .................................................... 13
2.5 Analysis of energy efficiency versus flexibility trade-off for a number of common architectural styles (for a 0.25 micron CMOS process)[1] ..................................................... 15

3.1 The Zigbee implementation system block diagram ......................... 17
3.2 Event driven architecture ........................................................................ 18
3.3 Event handler state machine ................................................................. 19
3.4 Event handler state machine ................................................................. 20
3.5 Zigbee frame structure ........................................................................... 25
3.6 The input data and output frame format of parsing and classification unit ..................................................... 25

4.1 The integrated Zigbee node ................................................................. 31
4.2 Celocia’s RC10 board and its block diagram .................................... 32
4.3 Clock module architecture ................................................................. 33
4.4 CC2420 evaluation board ................................................................. 34
4.5 Communication range and packet error rate test setup .................... 37
LIST OF FIGURES

4.6 Transmission distance vs. Packet error rate. The test nodes operate on 1.2MHz
system clock. RF frequency is 2405MHz. Radio transmitter output power is 0dBm
and antenna gain is 4.4dBi. .......................................................... 39

4.7 System throughput vs. Clock frequency .......................................... 43
List of Tables

1.1 Frequency band and data rate for IEEE 802.15.4 ........................................................... 3

2.1 Mica2 platform current draw measured with a 3V power supply .................................... 10

4.1 LEDs descriptions ................................................................................................................. 32
4.2 CC240 signals ........................................................................................................................... 34
4.3 The Zigbee node synthesis results ........................................................................................ 35
4.4 Functional test list .................................................................................................................... 37
4.5 The packet error rate of two zigbee nodes between 50cm ............................................... 41
4.6 The execution cycle counts of regular sensor network tasks among different approaches.
   The number in the table is in cycle ..................................................................................... 42
4.7 The FPGA usage comparison of hardware-based Zigbee node and MicroBlaze CPU . 44
### List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSMA-CA</td>
<td>Carrier Sense Multiple Access with Collision Avoidance</td>
</tr>
<tr>
<td>ED</td>
<td>Energy Detection</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FFD</td>
<td>Full-function device</td>
</tr>
<tr>
<td>GTS</td>
<td>Guaranteed Time Slots</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>kbps</td>
<td>kilo bits per second</td>
</tr>
<tr>
<td>LQI</td>
<td>Link Quality Indication</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>MPDU</td>
<td>MAC Protocol Data Unit</td>
</tr>
<tr>
<td>MSDU</td>
<td>MAC Service Data Unit</td>
</tr>
<tr>
<td>PER</td>
<td>Packet Error Rate</td>
</tr>
<tr>
<td>PHY</td>
<td>Physical Layer</td>
</tr>
<tr>
<td>PSDU</td>
<td>PHY Service Data Unit</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RFD</td>
<td>Reduced-function device</td>
</tr>
<tr>
<td>RF</td>
<td>Radio Frequency</td>
</tr>
<tr>
<td>RSSI</td>
<td>Receive Signal Strength Indicator</td>
</tr>
<tr>
<td>RX</td>
<td>Receive</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td>TX/RX</td>
<td>Transmit / Receive</td>
</tr>
<tr>
<td>TX</td>
<td>Transmit</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very high speed integrated circuit (VHSIC) hardware description language</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Network</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Wireless Sensor Networks

A wireless sensor network (WSN) consists of spatially distributed autonomous sensor nodes that communicate by wireless. These nodes collect data about physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants, and process the data and relay it to the end user. Wireless sensor networks intend to provide information that is precisely localized in time and/or space according to the user's needs or demands. Therefore, it enables a lot of applications from military target tracking, environmental monitoring, industrial control to home automation and health-care industries. These applications are expected to change the way we live, work and interact with the physical world [2] [3].

Wireless sensor networks have many advantages over traditional centralized sensing systems. They provide more localized and accurate information about monitoring targets since sensor devices are usually deployed in large scale. Wireless system is also easier to deploy and more scalable than wired system. The biggest advantages of wireless sensor network, however, are perhaps its
reliability and robustness which come from mesh network technology [4]. In a mesh network, sensor nodes need to reach only nearby nodes. Therefore, the transmission power and interference in radio signal are low. Consequently, link quality and reliability is increased. A mesh network offers robust communications among sensor nodes through its self configuring and self healing capabilities. A sensor node in a mesh network can directly connect to the nodes within its communication range and to all other nodes via multiple hop paths. The communication paths in mesh network are redundant. When there are broken or blocked paths, mesh networking allows for the reconfiguration. Therefore, communication is more robust in handling individual node or link failures.

There are some unique characteristics of a wireless sensor network [5] [6] [7]:

- Small node size: The size of sensor nodes in real world applications tends to be small to minimize the cost of sensor deployment.

- Limited power: Wireless sensor network nodes have very tight energy constraints. Most applications require sensor nodes to maintain operating condition for months or years on battery power.

- Large scale of deployment: Large scale of deployment increases the observation accuracy and reduces the communication distance between nodes. This also improves communication quality and increases energy efficiency. Large number of sensor nodes is also a prime enabler for robust communication in mesh wireless sensor network.

- Low Cost: Since sensor nodes are deployed in large scale, the cost of a single node is the dominating factor in the total cost of wireless sensor network applications. Also, most wireless sensor network applications in civilian domains are called price-enabled applications. This means that market waits for price-competitive devices to use the application.

- Unattended operation: Since large-number of devices are deployed in a wireless sensor network, unattended operation of each node and failure-tolerance operation becomes the basic requirement.

- Mobility: Sensor nodes may change their location after initial deployment. The mobility of
1. INTRODUCTION

nodes may be caused by incidental influence from environment, such wind or water. Or it is a desired property of system. For example, the nodes are attached to mobile entities.

- Heterogeneity: Sensor networks may consist of different types of nodes. Some types of nodes may have more computational power than others; The degree of heterogeneity in a sensor network affects the complexity and management of the whole system.

These unique characteristics of wireless sensor network make a rich source of research topics and educational activities. Several standards have been proposal based on specific technology and targeting specific application area. For example, IEEE 802.15.4 and ZigBee protocols.

The IEEE 802.15.4 [8] standard defines the protocol and interconnection of devices via radio communication in a low range wireless personal area network (LR-WPAN). The physical layer (PHY) and medium access control (MAC) layer specifications are defined in the standard. The IEEE802.15.4 standard intends to provide a standard for low complexity, low cost, low power consumption, and low data rate wireless connectivity among inexpensive devices. There are three frequency bands used in IEEE 802.15.4 standard. Sixteen channels are available in the 2450 MHz band, ten channels in the 915 MHz band, and one channel in the 868 MHz band. The modulation and spreading formats summarized in Table 1.1 [8].

Zigbee[9] is a low-cost, low power, two-way wireless sensor network communication standard developed by Zigbee Alliance, an organization of over 200 companies in 20 countries. It is targeted for low data rate applications, such as remote monitoring and control in home automation, manufacturing automation and industrial control. Zigbee is based on the IEEE 802.15.4 standard for

<table>
<thead>
<tr>
<th>PHY (MHz)</th>
<th>Frequency band (MHz)</th>
<th>Spreading parameters</th>
<th>Data parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Chip rate (kchips/s)</td>
<td>Modulation</td>
<td>Bit rate (kb/s)</td>
</tr>
<tr>
<td>868/915</td>
<td>300</td>
<td>BPSK</td>
<td>20</td>
</tr>
<tr>
<td>902-928</td>
<td>600</td>
<td>BPSK</td>
<td>40</td>
</tr>
<tr>
<td>2450</td>
<td>2000</td>
<td>O-PSK</td>
<td>250</td>
</tr>
</tbody>
</table>

Table 1.1: Frequency band and data rate for IEEE 802.15.4
1. INTRODUCTION

Zigbee Standard

Application Framework

Application Object n

Application Object 1

Zigbee Device Object

Application Support Sublayer (APS)

Network (NWK) Layer

Medium Access Control (MAC) Layer

Physical Layer

IEEE802.15.4 Standard

Figure 1.1: Zigbee protocol architecture

Wireless personal area networks (WPANs). It contains interface descriptions, object descriptions, protocols and algorithms. The stack architecture and the relation between IEEE802.15.4 and Zigbee are shown in Figure 1.1.

Wide variety of wireless sensor network applications have attracted many researchers to this area. Although wireless sensor nodes tend to be application specific because of a wide range of requirements and constraints in real-world sensor network applications, most current research still uses Mote [6] type sensor nodes, due to its availability and flexibility. Mote type sensor nodes were developed at the University California at Berkeley. Mote type sensor nodes are CPU-based embedded systems, which consist of a general purpose processor, memory, wireless communication subsystem and sensor subsystems. Because of the availability of general purpose CPU, dedicated operating system (TinyOS [10]) for sensor nodes and high level programming language tools, developing a new sensor network application on Motes is fast. That is why Berkeley motes are popular in academic
1. INTRODUCTION

research as well as for prototyping in commercial research.

The power consumption in Motes can be analyzed in three main parts: (1) CPU; (2) wireless communication subsystem; and (3) sensor subsystem. The power consumption of CPU depends on applications. In [11], it is studied that the power consumption of CPU varies from 20% to 80% of the total power consumption, but on average it is roughly 50%. A big fraction of the power consumed by the CPU is for processing the operating system routines and interrupts handling overhead. Therefore, the CPU-based sensor devices are not very energy efficient. Some researchers managed to decrease the energy consumption of CPU by introducing dedicated low power processors for the sensor network applications [12], while others propose using new event-driven architecture to improve energy efficiency of sensor devices[13][14].

1.2 Thesis Objective

Although there are studies [14][13] demonstrating that using hardware to leverage event-driven nature of sensor network applications is a possible way of improving energy efficiency, no hardware-based sensor nodes have yet been developed in the academic research for application testing because of the design time and high cost requirements in developing applications on customer hardware. Prototyping a hardware-based sensor node in FPGA is of interest because it allows testing hardware-based sensor nodes in real-world environment while reducing the development time and cost. The FPGA implementation also allows rapid design space exploration [15].

This thesis research has two objectives:

• (1) Prototyping a hardware-based sensor node on FPGA. To be application specific, the implementation is based on Zigbee standard. The FPGA is chosen as the implementation platform due to its low cost and flexibility.

• (2) Testing the implemented system in real-world environment. This test is not only to verify the system functionality, but also to measure the performance of the system.
1.3 Thesis Organization

The remainder of the thesis is organized as follows: Chapter 2 provides the research background and reviews related research in the design and implementation of sensor network nodes. Chapter 3 presents the architecture of the proposed system and describes the system components used. Chapter 4 presents implementation details targeting a Xilinx Spartan 3 FPGA. It also presents the simulation and experimental results obtained after testing the proposed sensor node in a wireless sensor network. Finally, we conclude in Chapter 5 with a discussion of future work.
Chapter 2

Background and Previous Work

In this chapter we review the research work that has been done for designing sensor network nodes. There are some fundamental requirements for nodes used in sensor networks [6] [16]:

- Small size and low power consumption: The processing, storage and interconnect capability of sensor devices is constrained by the size and power. Therefore the sensor devices must make efficient use of its computational unit and power.

- Event-driven and concurrency-intensive operation: The primary operation of sensor network device is transmit information from device to device. The information in wireless sensor network are simultaneously captured from sensor devices, processed, and sent or received to/from network through multi-hop routing. Therefore, tasks of sensor devices are interwoven with each other.

- Diversity in design and usage: Wireless sensor networks have a range of applications with vastly varying requirements and characteristics [17]. For example, active sensors, like sonar, require much more computational power for signal processing than passive sensors, such as smoke detection sensor. Sensors for applications that support mobility need more network
processing power to support complex network protocols and algorithms. That is why a single hardware platform is not sufficient to support such wide range of possible applications [18].

In order to address requirements described above, several design approaches are used in designing sensor network nodes. Generally, these approaches could be categorized into dedicated embedded sensor system, dedicated processor for sensor network application and dedicated architecture of sensor node.

2.1 Dedicated Embedded Sensor System

The dedicated embedded sensor system approach is to modify and optimize existed embedded system hardware and software to realize sensor device function and achieve energy efficient result. Figure 2.1 depicts a typical architecture of this kind of sensor node. The sensor node is a CPU-based system which consists of a general purpose micro-controller or processor with external memory, a wireless communication subsystem for communication with other nodes and a sensor module with a set of sensors. The general purpose processor is responsible for control and computation. There is a small, real-time operating system tailored for sensor network application running on the dedicated embedded system, such as TinyOS [10], Nano-Qplus[19] and Sensos [20]. TinyOS is an open source component-based operating system, which enables an efficient scheme of controlling hardware modules inside an embedded system. The system provides built-in interfaces, components, and sensor-board specific configurations. The size of operating system could be adjusted based on the complexity of hardware used in the sensor node. The adjustable feature is extremely important for sensor nodes since the memory size of sensor system is very limited. The general CPU-based architecture of dedicated embedded sensor system is a typical approach for sensor network device design in academic research. For example, the Berkeley Mote family[6], UCLA Medusa family[21] and MIT rMP[22] are based on the mote architecture.

The advantages of dedicated embedded sensor system are its flexibility and convenience in designing sensor network applications. The hardware platforms use off the shelf commodity IC chips. The operating system running on the system provides well defined software/hardware interface to
users, and general purpose CPU is supported by a high level (i.e., C, C++, and Java) program development environment. The available software tools and debugging platforms enable faster implementation process. For example, developers could program a sensor application in C language, compile it using available compiler, download and run it on the targeted embedded sensor system. The design flow is simple and the development is fast. Since algorithms are implemented in software, they can be easily modified, if necessary. That is why the Berkeley Mica family is so popular among academic researchers.

Although the dedicated sensor network operating systems, such as TinyOS, have already been tailored for small size and light weight, the operating routine and interrupt handling still contribute to a lot of workload for the CPU of embedded system. To illustrate the overhead, a broken-down cycle count for a simple application of Blinking LED in TinyOS is shown in Figure 2.2 [13]. The Blink LED application includes operations such as setting up a periodic timer, interrupting the system when time is up, queuing a blink LED function on the TinyOS task queue and executing the blinking LED program. The interrupt service and scheduler overhead consume 507 cycles while the blink LED task only consumes 16 cycles. The overhead is over 90 percent. Another example is execution cycles needed in MICA to complete a sequence of tasks: sampling sensor output, averaging the sampled data, and displaying the result. 781 cycles out of total 1118 cycles are used for interrupt handling and scheduling [13]. This implies roughly 70 percent overhead.

The TinyOS functions and interrupt handling overhead in dedicated embedded sensor system increase execution cycles, and consequently reduce the energy efficiency for any given application.
2. BACKGROUND AND PREVIOUS WORK

![Diagram: Dedicated Embedded Sensor Node System Architecture]

- **Timer Interrupt**
  - TinyOS Timer
  - Post BLINK to Task Queue
  - Execute BLINK
  - Go to sleep

- **Interrupt Service Routine**
  - 122 cycles

- **Software Timer**
  - 276 cycles

- **TinyOS Scheduler**
  - 88 cycles

- **Blink LED**
  - 16 cycles

- **TinyOS Scheduler**
  - 24 cycles

**Figure 2.2: Dedicated Embedded Sensor Node System Architecture**

<table>
<thead>
<tr>
<th>Device/Mode</th>
<th>Current</th>
<th>Device/Mode</th>
<th>Current</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td></td>
<td><strong>Radio</strong></td>
<td></td>
</tr>
<tr>
<td>Active</td>
<td>8.0mA</td>
<td>Rx</td>
<td>7.0mA</td>
</tr>
<tr>
<td>Idle</td>
<td>3.2mA</td>
<td>Tx(-20dBm)</td>
<td>3.7mA</td>
</tr>
<tr>
<td>ADC Acquire</td>
<td>1.0mA</td>
<td>Tx(-8dBm)</td>
<td>6.5mA</td>
</tr>
<tr>
<td>Extended Standby</td>
<td>0.223mA</td>
<td>Tx(-0dBm)</td>
<td>8.5mA</td>
</tr>
<tr>
<td>Standby</td>
<td>0.216mA</td>
<td>Tx(10dBm)</td>
<td>21.5mA</td>
</tr>
<tr>
<td>Power-save</td>
<td>0.110mA</td>
<td>Sensor</td>
<td></td>
</tr>
<tr>
<td>Power-down</td>
<td>0.103mA</td>
<td>Typical Board</td>
<td>0.7mA</td>
</tr>
</tbody>
</table>

**Table 2.1: Mica2 platform current draw measured with a 3V power supply**

Tasks. Therefore, the dedicated embedded sensor system is not very power efficient. Table 2.1 shows the measurement results for the power consumption of Mica2, a dedicated embedded sensor system, from PowerTOSSIM project[11]. The energy consumed has been broken down by different components. Table 2.1 shows that CPU, memory and radio module are the main power consuming components of the system. Studies on common tasks of sensor networks show that the power consumption of CPU in dedicated embedded sensor systems ranges from 28% to 86% of the total power and is roughly 50% on average. Since the operating system overhead in a dedicated embedded sensor system is significant, a large fraction of CPU power consumption is wasted in processing the overhead.

Latency is another issue for dedicated embedded system. CPU-based architecture has limited
computational parallelism. Concurrent events have to share common computational resources. This leads to increased latency of task processing. For real-time data acquisition applications that need high computation performance, systems either use a high-end processor, which lead to much higher power consumption, or adopt multiple CPU architecture [23], which increases system complexity.

2.2 Dedicated Processor for Sensor Network Application

Since general purpose processor in dedicated embedded sensor system consumes large fraction of the power, many researchers proposed reducing the power consumption of CPU to improve energy efficiency. Dedicated processors, such as asynchronous processor (SNAP/LE)[13] and the second-generation sensor processor [24], have been proposed for sensor network applications.

![SNAP architecture](image)

Figure 2.3: SNAP architecture

Dedicated sensor processors have specific instructions set architecture (ISA) for sensor network tasks. Unnecessary operations are removed from instruction set; and some new specific instructions for the operation of sensor nodes are added. In order to support the changes in instructions set, processor micro-architecture needs to be modified and new hardware modules need to be added to the new architecture. For example, the SNAP's [13] instruction set contains instructions that can be classified into five categories: (1) standard RISC(Reduced instruction set computer) instructions, (2)timer coprocessor instruction, (3) message coprocessor instructions, (4)network-protocol instruc-
2. BACKGROUND AND PREVIOUS WORK

The instruction set optimization results in a smaller program code size. The dedicated processor can also improve the scheduling function which makes it possible to get rid of the operating system overhead. It is reported in [13] that about 60% reduction of execution cycle count is obtained for executing sensor network tasks by using dedicated sensor network processor.

Dedicated sensor processor also minimizes the energy consumed for executing instructions using a wide variety of techniques. For example, subthreshold-voltage circuit technique is used in [12] and asynchronous circuit is used in [13]. These techniques usually come with a trade-off in performance. For example, subthreshold-voltage circuit technique results in slower clock speed[25].

2.3 Dedicated Architecture for Sensor Network Application

Dedicated architecture for sensor network application is an event-processor-based architecture proposed by Harvard University [14]. It is intended to improve the power efficiency by fully leveraging the event-driven nature of sensor network applications. The architecture is shown in Figure 2.4. This architecture is different from the CPU-based embedded system in the way that events are handled by a dedicated event processor. The event processor is essentially a programmable state machine to perform repetitive interrupt handling tasks. The hardware modules, such as message processor, data filter, assist the event processor. These modules are designed for specific tasks. Therefore, they improve power and cycle efficiency. The general purpose processor in the architecture is used as the last resort for computation. It is used only when other modules do not have the requisite function.

Wireless sensor network are inherently event-driven [26]. Dedicated embedded systems use interrupts to indicate an event that requires system’s attention. Therefore, the primary task of dedicated embedded sensor node is to handle timer and external interrupts [14]. A typical interrupt handling procedure of an embedded system is as follows: When the occurrence of an event is detected, an interrupt will be signaled to CPU. Upon receiving the interrupt signal, the CPU has to save current
execution state and data. Then, the CPU loads corresponding event handling program from memory and executes. After the execution is finished, the CPU has to restore its previous state and continue the execution. The operations for embedded system to store and restore its execution state before and after an interrupt is the overhead. The event driven architecture can avoid this overhead by using dedicated event processor. The function event processor coordinates others hardware function modules to respond to events. There is no software overhead for interrupt handling.

Since event-driven architecture avoids the overhead of running an operating system and interrupt handling, the hardware designed for specific tasks improves the system performance. The event-processor-based system\cite{14} uses about 10 times less clock cycles than dedicated embedded system. The reduction of cycle count makes the event-processor-based architecture runs for less time at the same clock frequency or enables it to run at a lower clock frequency and still execute a task in the required time. The power consumption of a CMOS system is comprised of two parts, static power consumption and dynamic power consumption. The dynamic power consumption is calculated by \ref{2.1}.

\begin{equation}
    P_{\text{dyn}} = \sum \alpha f CV^2
\end{equation}

\textit{Figure 2.4: Event-processor-based architecture}
Where $P_{dyn}$ is the total dynamic power consumption of the system, $a$ is a measure of switching activity, $f$ is the clock frequency, $C$ is the output capacitance of the node, and $V$ is the power supply voltage.

Running for less time or at a lower clock frequency reduces dynamic power and energy consumption. Another advantage of running at a lower clock frequency is that the processor can usually run at a lower supply voltage, further reducing both dynamic and static power dissipation and energy consumption.

2.4 Motivation of Our Work

Dedicated architecture for sensor network applications can significantly improve performance of sensor system [14]. However, to our knowledge, there are still no sensor network applications developed for dedicated architecture platforms. Lack of such platforms motivates us to implement a Zigbee sensor node based on dedicated architecture to verify the system functionality and performance using real-world applications.

The sensor node is based on an event-driven architecture. All system functions are implemented using dedicated hardware modules. The dedicated hardware designed for specific tasks is more power efficient than general purpose CPU for processing the same tasks [27]. Rabaey [1] compared energy efficiency against flexibility among embedded processor, ASIC, FPGA and dedicated hardware. As illustrated in Figure 2.5, the energy efficiency of dedicated hardware is about three orders of magnitude higher than embedded processors.

Our design is targeted for implementation using FPGAs for flexibility, reasonably high speed, and low cost. FPGA is very suitable for prototyping of sensor nodes because of its re-programmability. It is also an excellent platform for future design space exploration.

In the next chapter we describe the system architecture for the proposed sensor node.
Figure 2.5: Analysis of energy efficiency versus flexibility trade-off for a number of common architectural styles (for a 0.25 micron CMOS process)[1].
Chapter 3

System Architecture

Wireless sensor networks are inherently event-driven[26]. All sensor node designs mentioned in the previous chapter try to leverage the event-driven nature of sensor network applications to make efficient use of resources. For example, TinyOS, the operating system for dedicated embedded sensor system, has event-driven execution model; SNAP[13] has event-driven execution of instructions; dedicated architecture for sensor network application in [14] has dedicated event processor. All these research efforts suggest that energy efficiency can be obtained from optimizing the event handling process. Our design also adopts specific event driven architecture to improve system performance.

The system architecture is illustrated in Figure 3.1. A Zigbee application is implemented using six hardware units. The Zigbee application consist of three layers of communication: (1) physical MAC layer(IEEE 802.15.4); (2) network layer(Zigbee) and (3) application layer. The hardware units are cross-layer designed to minimize the communication among different layer functions. For example, parsing and classification unit spans across three layers. Its input is in physical layer frame while its output could be in MAC layer command or network layer command or application layer data. The connection between these hardware units needs dedicated links instead of one or more
buses. The dedicated links allows simultaneous communication among hardware units. The features of our design are described in the following sections.

3.1 Event-Driven Architecture

All function units in our design are event-driven systems, except parsing and classification unit. The general architecture of the event-driven system is shown in Figure 3.2. There are event handlers, event dispatchers, data bus, control signals, and state signals in the architecture. Event dispatcher is used to detect the occurrence of an event and activate the corresponding event handler to handle that event. An event could be internal, such as timer signal, or external, such as packet available in radio module buffer. Event handlers are dedicated hardware subsystems designed for specific events. They are activated by an event dispatcher in response to a specific event. Event handlers can generate other internal events. In this case, it activates another event handler to process the internally generated events. An event handler consists of event processing unit, local registers and shared registers. The event processing unit contains customized state machine and data path for processing a particular event. The state machine controls the event processing. The data path provides hardware needed for processing event information. Local registers are used for temporary data storage during event processing. They can be accessed only by the corresponding event handler.
3. SYSTEM ARCHITECTURE

The shared registers are used for exchanging event information. Information from event dispatchers is stored in shared registers. Both event handler and event dispatcher are able to access to the shared registers. The control signal is used to trigger the event handler to change its state from idle to active. The state signals indicate the state of event handlers to event dispatchers. The data bus is used to transfer event information from event dispatcher/event handler to event handler. The output ports in the architecture are driven by the event handlers, which are responsible for the communication between hardware blocks.

![Event Driven Architecture Diagram](image)

Figure 3.2: Event driven architecture

An event dispatcher is a state machine designed to monitor the input ports of hardware blocks, receive event information, decode event information and activate corresponding event handler. Figure 3.3 illustrates a simplified version of the actual state machine of an event dispatcher. The event dispatcher stay in Idle state after reset or power-on until an event occurs. When an event occurs, event dispatcher receives event information from input port, decodes event information, and lookups corresponding event handler to activate. If the event handler is in Idle state, the event dispatcher will activate the event handler. The activation requires three steps. First, the event dispatcher signals the event handler changing its state from Idle to JobAdd state. Then the event dispatcher receives event information from input port and sends it to the shared registers of activated event handler. Finally, after the information transaction is complete, the event dispatcher signals the event handler to leave JobAdd state and begin processing the task. After activation, the event handler returns to
the *Idle* state and continues to monitor the input interface. If the event handler is not in *Idle* state when an event dispatcher has detected an event, the event dispatcher will hold the communication on the input port and wait until the event handler is ready. Then the event dispatcher continues the activation as described above.

![Event handler state machine](image)

Figure 3.3: Event handler state machine

Each event handler is a small hardware processing unit with customized control logic and data path to perform a specific task. Figure 3.4 illustrates a simplified version state machine of an actual event handler. There are three common states in every event handler: (1) *Idle*, (2) *JobAdd* and (3) *JobStart*. All event handlers have same functionality in these states. The state change of event handler from *Idle* to *JobAdd*, from *JobAdd* to *JobStart* is solely controlled by the activator, which is either the event dispatcher or the event handler that is activating this event handler. The event handler stays in *Idle* after reset or power on. When in *Idle* state, the event handler is ready to accept a task. Event handler is signaled to change from *Idle* to *JobAdd* state after the event dispatcher has detected an event. The *JobAdd* state shows that the event handler is engaged to an event handling and receiving event information. When in this state, the event information is transferred from event dispatcher to event handler’s shared registers. The event handler does nothing in this state but waits until the transfer is complete. When the information transfer is complete, the event dispatcher signals event handler to change its state from *JobAdd* to *JobStart*. From this state, the event handler begins to run its specific logic to process the event. After the processing is complete,
event handler will return to the *Idle* state and wait for next event. Note that the event handlers do not provide preemption operation and there is no priority to access a particular event handler. They follow first-come-first-serve rule. When an event handler is busy, event dispatcher has to wait until it is ready.

![Event handler state machine](image)

**Figure 3.4: Event handler state machine**

The operation of the designed system is driven by events. An event processing procedure consists of three stages: (1) event detection, (2) handler activation, (3) event processing. In event detection phase, event dispatchers are in *Idle* state and monitor the input ports. When an event happens, the event handler decodes event and begins handler activation phase. In handler activation phase, event handler first checks the availability of corresponding event handler. If the corresponding event handler is idle, event dispatcher transfers event information to the event handler. Otherwise, the event dispatcher will wait until the event handler is ready. During this period, event handler is in *JobAdd* state and event dispatcher in *Activate* state or *Wait* state. After the information is transferred, the procedure enters event processing phase. The event dispatcher returns to idle state and event handler starts the task processing. The whole event processing phase lasts until event handler finishes its task. As described above, the event dispatcher is responsible for the event detection and the event handler is responsible for event processing. Therefore, the event detection phase and event processing phase could be overlapped. Also, because event handlers are able to work independently of each other, the event processing phases of different events can be overlapped, making it possible to process events in parallel.

In our event driven system, all computation resources needed for processing a particular event are available locally in event handlers. Therefore, event handlers do not need to signal an interrupt...
to request the computational resources for processing the task. As a result, there is no interrupt overhead in the our system.

3.2 Hardware Acceleration

The most frequent tasks of a sensor node are sending a packet, receiving a packet and relaying a packet. These activities involve operations such as parsing, classification, framing. In network processors, these operations are accelerated in hardware using pipelining and parallel processing [28] [29]. As shown in Figure 3.1, the designed system adopts the architectural ideas from network processors to accelerate packet operations. There are two information paths in the designed system: (1) fast path and (2) slow path. The function units in fast path deal with operations that are directly performed on packets, such as header modification, parsing, classification, packet framing. This path has customized data path for IEEE 802.15.4 frames and Zigbee packets. It guarantees that the throughput of the designed system is able to keep up with the line speed. The slow processing path deals with operations that are related to network management and upper layer application, such as routing protocol handling and routing table updates. In a network processor, those functions are implemented by the central processing units (CPUs). However, in our system, a set of specific hardware modules, event handlers, replace CPUs to accelerate the task processing.

3.3 Parallel Distributed Computation

Our system has two parallel data paths, fast path and slow path. Inside hardware units, each event handler and event dispatcher has its own data path and control logic. They also work in parallel. The parallel architecture in our system increases the efficiency of message processing and the system throughput. The computation capability is distributed among event handlers which enable the system to respond an event whenever it occurs. Implementation and testing results in chapter 4 show that the parallel distributed computation allows our system to respond faster to application needs.
There is a trade-off between size of circuit and performance. The parallel distributed com­putation architecture of our system has larger size circuit than other CPU-based architectures. In our implementation, we did not encounter restrictions on circuit size. As shown by the synthesis results in Chapter 4, our implementation uses about 51 percent of the target FPGA logic capacity. There is still plenty of logic capacity left unused. For implementations with very tight circuit size constraint, it is possible for the designed system to combine some event processing units into one common processing unit to reduce circuit size.

3.4 System Components

Before presenting the system components, a brief description of Zigbee nodes and their behavior is provided here. There are three type of Zigbee nodes: (1) coordinator, (2) router, and (3) end device. The coordinator is responsible for initiating and maintaining the network. It is the root of the network and there is only one coordinator in a Zigbee network. Router is responsible for moving data and control messages through the network. Router extends the coverage of the network. End device contains just enough functionality to talk to coordinator or router; it cannot relay data from other nodes. Both coordinator and router provide an association service for new node to join to a network. Association is a procedure used to establish membership in a network. A parent-child relationship is formed when a node having membership in the network allows a new node to join. The new node becomes the child, while the other node becomes the parent. The parent will assign a 16-bit network (NWK) address to its new children based on its capability. For router, the 16-bit NWK address is calculated based on Equation 3.1; for end device, the function is determined by the Equation 3.2;

\[ A_{Router,n} = A_{parent} + Cskip(d) \cdot n \]  
\[ A_{EndDevice,n} = A_{parent} + Cskip(d) \cdot R_m + n \]
Where

\[ C_{skip}(d) = \frac{1 + C_m - R_m - C_m * R_m^{L_m - d - 1}}{1 - R_m} \]  

(3.3)

\( A_{\text{Router}_n} \) is the network address of \( n^{th} \) router's child. \( A_{\text{parent}} \) is the network address of the parent. \( C_{skip}(d) \) essentially the size of the address sub-block being distributed by each parent at that depth to its router-capable child node for a given network depth. It is given by Equation 3.3. \( C_m \) gives the maximum number of children a parent may have. \( L_m \) is the maximum depth of the network. It specifies the maximum number of allowable levels in a particular network tree. \( R_m \) is the maximum number of routers that a parent may have as children.

The implementation in this thesis support Zigbee coordinator functions and router functions. As shown in Figure 3.1, our system consists of parsing and classification unit, modification and framing unit, network control unit, application and management unit, radio control unit and CC2420 unit. System's functional units are connected to each other by dedicated links. The functions of these units are described below.

### 3.4.1 Radio control unit

Radio control unit interfaces to the CC2420 transceiver module to provide a communication channel to other modules in the system. It provides three operations: (1) get-packet, (2) send-packet and (3) TRX-configuration.

1. Get-packet: This operation is triggered when CC2420 module signals that there is a packet available in its buffer. Get-packet operation gets packets from CC2420 buffer and sends the packets to parsing and classification unit.

2. Send-packet: This operation is triggered when modification and framing unit sends a packet to radio control unit. Radio control unit receives the packet and sends it to CC2420 module. The whole send-packet operation consists of four steps. First, radio control unit will issue a clear command to CC2420 module to clear its buffer. Then, radio control unit receives the packet from modification and framing unit and sends it to CC240 buffer. After the packet
3. SYST E M ARCHITE C TU R E

transfer is finished, radio control unit will issue a send command to CC2420 module. When CC2420 receives this command, it changes from receive mode to TX mode. CC2420 will start to send the packet when the wireless channel is free. After sending the send command to CC2420, radio control unit keeps monitoring the state of CC2420 module until CC2420 finishes sending. When the send packet operation is finished the radio control unit return to idle state.

3. TRX-configuration: This operation is triggered when network control unit needs to command or configure CC2420 module, or read CC2420 module state. When network control unit sends command or configuration parameter to radio control unit, radio control unit will write them to the corresponding CC2420 register. At the same time, radio control unit reads CC2420 state and sends it back to network control unit. When network control unit reads the content of CC2420 register, radio control unit will send read-register command and the register address to CC2420 module. Then it receives the module’s states and the register’s value from CC2420 and sends them to network control unit.

3.4.2 Parsing and classification unit

Parsing and classification unit receive packets from radio control unit and parses the received packets into information fields. The parsed information is then sent to corresponding module for further processing based on the packet type. Some unnecessary information fields are removed. For example, the MAC command is sent to network control unit and the application data are sent to application and management unit. The destination MAC address is removed because it is not needed in further processing. The Zigbee frame structure is shown in Figure 3.5 and the input frame and output data format of parsing and classification unit are shown in Figure 3.6. The input frame types include beacon, acknowledgement, MAC command and MAC data.
3. SYSTEM ARCHITECTURE

Figure 3.5: Zigbee frame structure

Figure 3.6: The input data and output frame format of parsing and classification unit
3.4.3 Network control unit

Network control unit processes MAC layer and Zigbee network layer events. Its functions includes network discovery, network start, network join, neighbor table maintenance, etc. It is also responsible for configuring CC2420 transceiver, and resetting MAC layer and physical layer.

Network control unit interfaces to parsing and classification unit, application and management unit, modification and framing unit and radio control unit. In network control unit, there are two event dispatchers and fourteen event handlers. One event dispatcher monitors the interface between parsing and classification unit; the other monitors the interface between application and management unit. These two dispatchers work independently in parallel. The name of event handlers and their functions are listed as follows.

1. send.ToFrame: This event handler is activated when an event handler needs to send information to other nodes in the Zigbee network. The process will send data to modification and framing unit, where data are packed as a Zigbee frame.

2. send.ToNWK: This event handler is activated when an event handler needs to send information to application and management unit. This event handler is responsible for the communication between network control unit and application and management unit.

3. ToCC2420.process: This event handler is activated when an event handler needs to command or configure CC2420 module. This event handler is responsible for communication between network control unit and radio control unit.

4. beacon.process: This process is activated when network control unit receive a beacon. It compares the (personal area network Identifier) PAN ID in beacon with those in its neighbor table. If the PAN ID in beacon is different from those in neighbor table and neighbor table has still has capacity, it adds a new neighbor to its neighbor table and notifies application and management unit that a new neighbor is found. Otherwise, it ignores the beacon. In order to notify application and management unit, a new event process, send.ToNWK, will be activated.
5. associate_req: This process is activated when network control unit receives the MAC command, association request. It checks if the requesting nodes are already in children table. If they are, then event handler assigns the old 16-bit NWK address to the requesting node. If the requesting node is not in children table, event handler will assign a new 16-bit NWK address to the requesting node and adds new entity in children table. In both cases, the association attempt succeeds. If there is no capacity in children table, the association attempt fails. Event handler will send an association response command to tell the requesting node the result of association attempt. The association state and 16-bit NWK address will be in that command. In order to send the association response command, the send_ToFrame event handler will be activated at the end of this process.

6. associate_resp: This process is activated when network control unit receives the MAC command frame of association response. If the node has already been associated with a Zigbee network, event handler will ignore this event. If the association response is with a success state, the event handler will setup 16-bit network address and PAN ID based on the information from association response command. It also activates ToCC2420_process event handler to configure CC2420 module, and activates send_ToNWK event handler to inform application and management unit that association with a Zigbee Network is successful. If the association response returns with a failure state, the event handler informs application and management unit that the association attempt failed.

7. beacon_req: This process is activated when network control unit receives a search for Zigbee network request from application and management unit. The event handler will construct a beacon request command and activate send_ToFrame event handler to send the command.

8. join_network: This event handler is activated when network control unit receives a join to a network command from application and management unit. The event handler will first lookup neighbor table to found a suitable Zigbee network to join. If no Zigbee network found, event handler will send a message back to application and management unit to inform that no network is available. If there is a suitable Zigbee network, event handler will send a
association request command to the network. The send.ToFrame event handler will be activated to send the Zigbee frame.

9. search_network: This event handler is activated when network control unit receive a search network command from application and management unit. A beacon request command will be constructed and broadcasted. The send.ToFrame event handler will be activated to send the Zigbee command.

10. send_NWKdata: This event handler is activated when network control unit receive a data package from application and management unit. Event handler constructs a Zigbee data frame. Frame length, frame control field, PAN ID, sequence number and source address are added to the frame. Then event handler activates the send.ToFrame event handler to send the frame.

11. initial_CC2420: This event handler is activated after sensor node is powered up or reset. The event handler sets up PAN ID, 64-bit IEEE address, 16-bit network address, frequency and buffer size in CC2420 module, It also turns on the crystal oscillator in CC2420 module and sets CC2420 in receive mode.

12. start_NWK: This event handler is activated when network control unit receive a start network command from application and management unit. Event handler sets up sensor node to be a Zigbee coordinator.

3.4.4 Application and management unit

Application and management unit interfaces to parsing and classification unit, network control unit, modification and framing unite. It has network management, application function and user interface function. The network management controls the process of setup and joining a Zigbee network and routing. Application function controls a test application. The test application mimics data monitoring scenario in real-world Zigbee application. User interface controls the output of LEDs and LCDs on Zigbee node, which are used in testing and debugging.
3. SYSTEM ARCHITECTURE

3.4.5 Modification and framing unite

Modification and framing unit is responsible for packing the data into frames that will be sent to other Zigbee nodes. It interfaces with parsing and classification unit, network control unit, radio control unit. It consists of a dispatcher and two event handlers. The dispatcher monitors the events at its two interfaces. If a packet is received from parsing and classification unit, event handler starts route lookup process. The route lookup process sends a request to application and management unit for next-hop address. The return address will be framed in the packet. For packets from network control unit, event handler adds MAC source address and sequential number. The frame is then transferred to radio control unit.

3.4.6 CC2420 module

CC2420 module is the low power, IEEE 802.15.4 [8] and ZigBee compatible, RF chip CC2420 [30]. It is used as a wireless transceiver. The description and functionality of this chip are presented in Chapter 4.
Chapter 4

Implementation and Results

4.1 Implementation Platform

Two Zigbee nodes were implemented in our work using Celoxica RC10 FPGA development boards[31] and Chipcon CC2420 evaluation boards (CC2420EB) [32]. The CC2420 evaluation boards are used as the transceiver unit of Zigbee node. The FPGA in RC10 board is used to implement parsing and classification unit, modification and framing unit, network control unit, application and management unit and radio control unit. The integrated Zigbee node is shown in Figure 4.1. The description of these prototyping boards is given in the following subsection.

4.1.1 RC10 development board

The RC10 FPGA evaluation board from Celoxica is designed for rapid prototyping and application development. RC10 contains a Xilinx Spartan 3 FPGA, computer interfaces, standard I/O, LEDs and LCDs display, and external circuits that support the operation of FPGA. The RC10 board and its function blocks are shown in Figure 4.2. The components used in the Zigbee node implementation
include eight LEDs, two LCDs and a Joystick. The LEDs are used to indicate the working state of the Zigbee node. The information displayed using these LEDs is described in Table 4.1. The two seven segment displays are used in testing applications to display the number of packets received by the node or the data value of the packets. The test applications are implemented in the Zigbee node to test the system functionality and performance. The Joystick is used as a push button to reset the system.

The FPGA on Celoxica RC10 development boards is Xilinx Spartan3 xc3s1500-fg320-4 which contains 29,952 logic cells. Each logic cell has a 4 input look-up table and a D flip-flop. The system capacity of the FPGA is equivalent to 1.5 million logic gates.

The system clock of FPGA is provided by RC10 Board. The clock frequency is fixed at 48.00 MHz. For the Zigbee implementation, a much slower clock is used. The 48 MHz board clock needs to be slowed down. There are four digital clock managers (DCM) in Spartan-3 devices available for clock control. DCM is able to divide an input clock by a factor of 1.5, 2.0, 2.5, 3.0, 4.0, 5.0, 8.0 or 16. In order to obtain a wider range of frequency control, two DCMs are connected in serial, as illustrated in Figure 4.3. These two DCMs work as a clock management unit. The input of clock
4. IMPLEMENTATION AND RESULTS

(a) RC10 (b) RC10 Block Diagram

Figure 4.2: Celcia's RC10 board and its block diagram

<table>
<thead>
<tr>
<th>LED Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LED0</td>
<td>For Zigbee coordinator or router. On when the node has detected that there is a new node looking for Zigbee networks</td>
</tr>
<tr>
<td>LED1</td>
<td>For nodes looking for Zigbee network. On when a Zigbee network is found</td>
</tr>
<tr>
<td>LED2</td>
<td>For nodes wanting to join a Zigbee network. On when a join network request is sent</td>
</tr>
<tr>
<td>LED3</td>
<td>For nodes looking for Zigbee network. On when the node send out a beacon request to actively search for existing Zigbee networks</td>
</tr>
<tr>
<td>LED4</td>
<td>For Zigbee coordinator or router. On when it accept a new node into network</td>
</tr>
<tr>
<td>LED5</td>
<td>For nodes wanting to join a Zigbee network. On when it has joined an existing zigbee network</td>
</tr>
<tr>
<td>LED6</td>
<td>For event. On When an event of sending application data occur</td>
</tr>
<tr>
<td>LED7</td>
<td>For Zigbee coordinator. On when it has started a new zigbee network</td>
</tr>
</tbody>
</table>

Table 4.1: LEDs descriptions
management unite is 48 MHz clock from RC10 board. Its output is a slower clock, the frequency of which could be configured through a slowdown factor ranging from 1.5 to 40. Since the minimum input clock frequency of DCM is 18MHz, the maximum slowdown factor of first DCM is 2.5. The output clock frequency, $F_{Output\_clock}$, is shown in Equation 4.1. By changing these slowdown factors, system performance for different frequencies can be measured.

\[
F_{Output\_clock} = \frac{48}{DCM1\_Slowdown \times DCM2\_Slowdown}
\]

(4.1)

where $DCM1\_Slowdown \in \{1.5, 2, 2.5\}$ and $DCM2\_Slowdown \in \{1.5, 2, 2.5, 3, 4, 5, 8, 16\}$

4.1.2 CC2400 evaluation board and CC2420

The CC2420 evaluation board (CC2420 EB) [32] is a development board from Chipcon. As shown in Figure 4.4, the board contains a CC2420 chip and external circuitry required for the operation of CC2420, such as power and oscillator circuit. The CC2420 EM enables users to use CC2420 chip without spending time to make a printed circuit board. All CC2420 digital pins are accessible from its test port 1. In our Zigbee node implementation, test port 1 is used to connect FPGA with CC2420 chip.

CC2420 is a single-chip 2.4 GHz IEEE 802.15.4 compliant RF transceiver designed by Chipcon.
4. IMPLEMENTATION AND RESULTS

Figure 4.4: CC2420 evaluation board

<table>
<thead>
<tr>
<th>Signal Category</th>
<th>Signal Name</th>
<th>Test Port 1 Pin Number</th>
<th>Signal Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI signals</td>
<td>CSn</td>
<td>12</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td>SCLK</td>
<td>13</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td>SI</td>
<td>14</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td>SO</td>
<td>15</td>
<td>output</td>
</tr>
<tr>
<td>CC2420 State Signals</td>
<td>FIFOP</td>
<td>4</td>
<td>output</td>
</tr>
<tr>
<td></td>
<td>SFD</td>
<td>8</td>
<td>output</td>
</tr>
<tr>
<td></td>
<td>CCA</td>
<td>16</td>
<td>output</td>
</tr>
<tr>
<td></td>
<td>FIFO</td>
<td>17</td>
<td>output</td>
</tr>
<tr>
<td>Other Signals</td>
<td>Resctn</td>
<td>18</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td>VREG</td>
<td>19</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td>GND (ground)</td>
<td>20</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 4.2: CC2420 signals

Its output power is programmable from -24 dBm to 0 dBm and its receiver sensitivity is -95 dBm.

The signals of CC2420 that need to be connected to the FPGA are shown in Table 4.2. The Serial Peripheral Interface (SPI) signals are used for CC2420 configuration and for accessing the chip’s buffer. The state signals are output signals from CC2420 to indicate the transceiver’s working states. The functional description of SPI and CC2420 state signals can be found in [32]. Reset signal is the same signal that resets the whole system (sensor node). VREG (voltage regulator) signal should always high to enable the regulator. The ground pins of two development board are connected together to make the signals of two boards have the same reference voltage.
4. IMPLEMENTATION AND RESULTS

<table>
<thead>
<tr>
<th>Resource type</th>
<th>Slices</th>
<th>Slice Flip-Flops</th>
<th>4 Input LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available</td>
<td>13312</td>
<td>26624</td>
<td>26624</td>
</tr>
<tr>
<td>Parsing and Classification Unit</td>
<td>Used</td>
<td>3180</td>
<td>1223</td>
</tr>
<tr>
<td>Percent</td>
<td>23%</td>
<td>4%</td>
<td>22%</td>
</tr>
<tr>
<td>Modification and Framing Unit</td>
<td>Used</td>
<td>374</td>
<td>315</td>
</tr>
<tr>
<td>Percent</td>
<td>2%</td>
<td>1%</td>
<td>2%</td>
</tr>
<tr>
<td>Application and Management Unit</td>
<td>Used</td>
<td>268</td>
<td>231</td>
</tr>
<tr>
<td>Percent</td>
<td>2%</td>
<td>0%</td>
<td>1%</td>
</tr>
<tr>
<td>Network Control Unit</td>
<td>Used</td>
<td>2397</td>
<td>1901</td>
</tr>
<tr>
<td>Percent</td>
<td>18%</td>
<td>7%</td>
<td>16%</td>
</tr>
<tr>
<td>Radio Control unit</td>
<td>Used</td>
<td>326</td>
<td>170</td>
</tr>
<tr>
<td>Percent</td>
<td>2%</td>
<td>0%</td>
<td>2%</td>
</tr>
<tr>
<td>Total System</td>
<td>Used</td>
<td>6888</td>
<td>3549</td>
</tr>
<tr>
<td>Percent</td>
<td>52%</td>
<td>13%</td>
<td>49%</td>
</tr>
</tbody>
</table>

Table 4.3: The Zigbee node synthesis results

4.2 Implementation Details

The Zigbee node design and implementation was done using VHDL [33], synthesized using Xilinx ISE7.0 [34] and simulated using Mentor Modelsim simulator [35]. The synthesis results are shown in Table 4.3. The sensor node system uses about 52 percent of FPGA resources. It means that the size of implemented system is about 700,000 gates. The parsing and classification unit, and network control unit are the two largest units in the implemented system, which consist of about 80 percent of system logic, while application and management unit only take up 2 percent of system logic in our implementation. However, if a more complex application or sophisticated routing algorithm is implemented in the sensor system, application and management unit will be larger.

4.3 Test Results

In order to evaluate the performance of implemented system, the following parameters were tested: range, packet error rate, execution cycle count and throughput. Actual hardware was used to measure the range and packet error rate. Execution cycle count and throughput were estimated using ModelSim simulator.

An application was developed to test range, packet error rate and system functionality. This test application mimics data monitoring scenario in real-world Zigbee applications. The sensor node...
periodically collects sampled data and transmits packets containing the data to its remote peer. The remote peer receives packets, processes them and displays the data value.

4.3.1 Functional tests

Two Zigbee nodes were used in the functional test. The whole testing process was divided into three stages. In the first stage, a Zigbee node sets up a Zigbee network. The Zigbee node performs system initializing, network searching and network setup functions. It works as a Zigbee coordinator. In the second stage, the other Zigbee node joins the Zigbee network. It finds and joins the Zigbee network set up by the first Zigbee node. It works as a Zigbee router. In the third stage, communication between these two nodes is tested. The Zigbee router collects data from its internal counter periodically and sends it to the Zigbee coordinator. The Zigbee coordinator receives the data packets from Zigbee router, processes them and displays the data value on 7-segment displays. The verification of the system functions is done by reading the state indicators and LCDs on the RC10 FPGA board.

The state indicators are the LEDs on RC10 board, which are turned on when a specific function has been successfully accomplished. The meaning of each LED is shown in Table 4.1. The data received by Zigbee coordinator is displayed on the 7-segment LCDs. Since the data to be sent is collected from a counter, the data are known. If the data shown on LCDs are same as what we expected, the data receiving function and data sending function are verified.

In functional test, two Zigbee nodes were able to setup a Zigbee network and communicate with each other. The nodes displayed the expected values which confirmed that the functional tests successfully passed. The functional test results are shown in Table 4.4. The test results demonstrate that the nodes able to function as a Zigbee coordinator or a Zigbee router in the Zigbee network.

Note that there are a few functions that could not be tested on hardware platform due to the limit on number of nodes (only two nodes were available). Those functions were verified using simulator. The functions validated using simulator include the packet relay function, join network refusal.
4. IMPLEMENTATION AND RESULTS

<table>
<thead>
<tr>
<th>Tested Function</th>
<th>Function Description</th>
<th>Test Platform</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>System initial</td>
<td>Initial internal counter, network address and CC2420</td>
<td>Hardware</td>
<td>Pass</td>
</tr>
<tr>
<td>Network search</td>
<td>Request beacon, receive and decode network information from beacon and update network candidate list</td>
<td>Hardware</td>
<td>Pass</td>
</tr>
<tr>
<td>Network setup</td>
<td>Start a Zigbee network</td>
<td>Hardware</td>
<td>Pass</td>
</tr>
<tr>
<td>Beacon broadcast</td>
<td>Broadcast beacon that contains network information</td>
<td>Hardware</td>
<td>Pass</td>
</tr>
<tr>
<td>Network join request</td>
<td>Choose a network from candidate list and send out associate request command</td>
<td>Hardware</td>
<td>Pass</td>
</tr>
<tr>
<td>Network join</td>
<td>Receive respond from target network, update its PAN ID, network address and initial Zigbee router function if join network is succeeded</td>
<td>Hardware</td>
<td>Pass</td>
</tr>
<tr>
<td>Accept a node</td>
<td>Receive associate request, check its capacity, assign address and update son table if a new node is allowed to join, send out associate respond</td>
<td>Hardware</td>
<td>Pass</td>
</tr>
<tr>
<td>Send Data</td>
<td>Send an application data to peer node</td>
<td>Hardware</td>
<td>Pass</td>
</tr>
<tr>
<td>Receive Data</td>
<td>Receive a packet from remote peer</td>
<td>Hardware</td>
<td>Pass</td>
</tr>
<tr>
<td>Relay data</td>
<td>Receive a packet and relay to next-hop node</td>
<td>Simulator</td>
<td>Pass</td>
</tr>
<tr>
<td>Reject a node</td>
<td>Receive associate request, check its capacity, send out associate respond with rejection if a new node is not allowed to join</td>
<td>Simulator</td>
<td>Pass</td>
</tr>
</tbody>
</table>

Table 4.4: Functional test list

![Communication range and packet error rate test setup](image)

Figure 4.5: Communication range and packet error rate test setup

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
4. IMPLEMENTATION AND RESULTS

4.3.2 Packet error rate versus communication range

The effect of change in communication range on packet error rate was determined using the two implemented Zigbee nodes. The objectives of this experiment were: first, to measure the working range of the implemented nodes. The second, to measure the packet error rate when the communication range was changed. The test setup is shown in Figure 4.5. Two Zigbee nodes are used in the test. One is for transmitting data and the other is for receiving data. The sending node records the total number of packets sent, $P_{\text{sent}}$; and the receiving node records the total number of packets received, $P_{\text{received}}$. Note that the packets counted at the receiving node are the packets received at application layer. There are packets with error received by Zigbee node. These packets will be detected by cyclic redundancy check (CRC) and dropped off at MAC layer. The packet error rate ($PER$) is given by:

$$PER = \frac{(P_{\text{sent}} - P_{\text{received}})}{P_{\text{sent}}}$$

The tests were conducted in both indoor and outdoor environments. The test results are shown in Figure 4.6. As illustrated in the figure, there are three communication zones. The 0~30 meters in the indoor environment and 0~65 meters in the outdoor environment are good communication zones. The packet error rate in these zones is below 2%. Within this range, the implemented system is able to provide reliable operation. The 30~55 meters in the indoor environment and 65~85 meters in the outdoor environment are distorted communication zones. The packet error rate there is from 5% to 20%. The implemented system is still able to function in this zone, but the communication is unreliable. More than 55 meters in the indoor environment and 85 meters in the outdoor environment is out of service zone. The implementation has a hard time setting up the network and establishing a communication link.

4.3.3 Crosstalk induced packet error

Packet Error Rate (PER) is an important quality of service parameter (QoS) for wireless network. It is closely related to the noise and interference in the communication channel. In the packet error rate test described above, the system suffers from three different interference sources. One is
Figure 4.6: Transmission distance vs. Packet error rate. The test nodes operate on 1.2MHz system clock. RF frequency is 2405MHz. Radio transmitter output power is 0dBm and antenna gain is 4.4dBi.
the background noise which is a common noise produced by electronic nodes. The noise signal is uncorrelated in time and its power spectral density is flat within the given communication channel's bandwidth. A good approximation of this noise is Gaussian white noise. Another interference source is wireless LAN (WLAN). The IEEE 802.15.4 (Zigbee) standard uses the 2.4GHz industrial scientific and medical (ISM) unlicensed band which is also used by IEEE 802.11b (WLAN) and IEEE 802.15.1 (Bluetooth). Since there is campus-wide WLAN coverage, the test systems suffer interference from WLAN. The third interference comes from the connection cables that connect RC10 board and CC2420EB board. These cables induces crosstalk noise [36] in the circuit of the test system. The noise frequency is low, about several megahertz.

The white noise and WLAN's interference in the Zigbee communication channel is common for most real-world Zigbee applications. The cross talk noise in our test system, on the other hand, is unique. This noise is generated by inferior wire connection in the implementation; and can be decreased or eliminated by placing all components in a printed circuit board with well designed layout. Therefore, it is important to investigate and isolate the packet error rate introduced by the connecting cables.

The channel width of WLAN (802.11b) is 20MHz, while the channel width of Zigbee is 5Mhz. Since WLAN channel is much wider than Zigbee, the effect of WLAN signal is flatted in Zigbee channel. Therefore, the WLAN interference to Zigbee node can be approximated as white noise [37]. White noise is additive, so the effect of WLAN signal and background noise can be considered together as white noise effect. How white noise affects bit error rate in Zigbee is well studied. When signal to noise ratio (SNR) is high, the noise effect on bit error rate is small. For example, when SNR is 5dB, the BER is about $10^{-3.8}$ [38]. So if the packet error rate is measured in high SNR situation, this packet error rate could be approximated as the packet error rate induced by jumping cables. In order to estimate the effect of jumping cables, Two Zigbee nodes are placed within 50cm to test the packet error rate. The test result is shown in Table 4.5.

As shown in Table 4.5, the packet error rate is about 1% when two nodes are very close to each other. It is fair to say that if the connection between FPGA and CC2420 transceiver is improved, the packet error rate will be improved at least 1% for the communication between nodes in short
### 4. IMPLEMENTATION AND RESULTS

<table>
<thead>
<tr>
<th>Test Number</th>
<th>Number of Packet Sent</th>
<th>Number Packet Received</th>
<th>Packet Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>65535</td>
<td>64996</td>
<td>0.008292818</td>
</tr>
<tr>
<td>2</td>
<td>32768</td>
<td>32468</td>
<td>0.009239867</td>
</tr>
<tr>
<td>3</td>
<td>32768</td>
<td>32730</td>
<td>0.001161014</td>
</tr>
<tr>
<td>4</td>
<td>32768</td>
<td>32579</td>
<td>0.005801283</td>
</tr>
<tr>
<td>5</td>
<td>32768</td>
<td>32534</td>
<td>0.007192176</td>
</tr>
<tr>
<td>6</td>
<td>32768</td>
<td>32437</td>
<td>0.010204396</td>
</tr>
<tr>
<td>7</td>
<td>8192</td>
<td>8127</td>
<td>0.007999031</td>
</tr>
<tr>
<td>Total</td>
<td>237567</td>
<td>235871</td>
<td>0.007190371</td>
</tr>
</tbody>
</table>

Table 4.5: The packet error rate of two Zigbee nodes between 50cm distance. The short distance here means the signal to white noise rate is high enough to overlook the packet error rate caused by white noise. For communication in long distance, the improvement is unpredictable based on this test because the effect of connecting cable on packet error rate cannot be isolated from those of other interference sources.

#### 4.3.4 Performance analysis using execution cycle count

To evaluate the efficiency of the hardware-based implementation, the designed system is compared against typical sensor node design approaches. The design approaches under evaluation include general purpose CPU solution[39], sensor-processor solution[13], event-processor-based solution[14] and our hardware-based solution. Execution cycle counts for regular sensor network tasks are used as performance indices for comparison. The need for fewer execution cycles means that system can either run for less time at the same clock frequency or can run at a lower clock frequency and still execute a task in the required time. Completing a given task in shorter time or lower clock frequency reduces dynamic power and energy consumption. In addition, running a system at a lower clock frequency enables lower supply voltage, further reducing power dissipation. The regular sensor network tasks used to measure the cycle counts in the evaluation are described as follows:

1) Packet transmission: The application and management unit periodically collects samples and sends sampled values to the network control unit, which decides the destination address for the packet and looks up next-hop address. All the packets are sent to modification and framing unit to form a MAC frame. The frame is then transferred to radio control unit and transmitted.

2) Packet reception: The radio control unit fetches the MAC frame from CC2420 buffer when it...
4. IMPLEMENTATION AND RESULTS

<table>
<thead>
<tr>
<th>Task</th>
<th>Mica2</th>
<th>SNAP</th>
<th>Event-processor-based system</th>
<th>Hardware-based system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Transm</td>
<td>1532</td>
<td>331</td>
<td>127</td>
<td>137</td>
</tr>
<tr>
<td>Packet Recpt</td>
<td>234</td>
<td>258</td>
<td>136</td>
<td>71</td>
</tr>
<tr>
<td>Packet Relay</td>
<td>429</td>
<td>418</td>
<td>165</td>
<td>115</td>
</tr>
</tbody>
</table>

Table 4.6: The execution cycle counts of regular sensor network tasks among different approaches. The number in the table is in cycle

is ready and sends the frame to parsing and classification unit. If the destination of the packet is for the sensor node, content parsed from frame will be sent to network control unit or application and management unit.

3) Packet relay: After the reception of packets, if the packet is not for the sensor node, the system will modify corresponding fields of the frame send it to next-hop node or destination node.

The execution cycle counts of different implementations for performing the regular sensor network tasks explained above are presented in Table 4.6. The first column shows the task type under evaluation. The second column shows the cycle counts of MICA2 which are obtained from [14]. The third column shows cycle counts of SNAP from [13]. The fourth column shows cycle count of event-processor-based architecture from [14]. The fifth column shows the cycle count of the implemented system which is determined using Mentor ModelSim simulator.

As shown in Table 4.6, the implemented system is about 11 times faster than Mica2 system in packet transmission task, and about 3~4 times faster in packet reception task and packet relay task. The cycle count comparison on typical tasks shows our system provides significant performance improvement over the general processor solution. This improvement is not only the result of avoiding the overhead of operating system and interrupt handling, but also from utilization of parallel processing. For example, in packet relay task, framing unit will start a route lookup event as soon as it receives the destination address of the relay packet. The framing event handler and route lookup event handler are able to run in parallel. The packet relay performance of our system is the best among all evaluated systems. Compared to the event-processor-based architecture, the implemented system uses 30% less cycle count.
4. IMPLEMENTATION AND RESULTS

300000
250000

Packet transmission
Packet Reception
Pack Relay

150000
100000

100
60 120

Frame Length

20 40 60 80 100 120

230 KHz, while keeping up with line throughput of 250K bit/s. The length of packets has little influence to the relay performance.

The performance results demonstrate that in one second, a hardware oriented Zigbee coordinator working at 300 KHz system clock is capable of processing 400 incoming data packets, relaying 200 packets and transmitting 10 control command. Therefore a Zigbee end device working at 300 KHz clock could have a sample rate of 1000 samples/second.

4.3.6 The trade-off of hardware-based system

Although our hardware-based implementation attains high system throughput and execution efficiency, the circuit size of our implementation is large. There is a trade-off between performance and circuit size. To illustrate this trade-off, a Xilinx soft core CPU, MicroBlaze, was implemented in FPGA as a reference design of CPU based sensor node. MicroBlaze is a 32-bit soft processor with
### IMPLEMENTATION AND RESULTS

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>Available</th>
<th>Hardware-based node</th>
<th>Zigbee</th>
<th>MicroBlaze CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>13312</td>
<td>1665</td>
<td>52%</td>
<td>13%</td>
</tr>
<tr>
<td>Slice Flip-Flops</td>
<td>26624</td>
<td>1248</td>
<td>13%</td>
<td>5%</td>
</tr>
<tr>
<td>4 Input LUTs</td>
<td>26624</td>
<td>2247</td>
<td>49%</td>
<td>8%</td>
</tr>
<tr>
<td>Bonded IOBs</td>
<td>221</td>
<td>81</td>
<td>16%</td>
<td>37%</td>
</tr>
<tr>
<td>GCLKs</td>
<td>8</td>
<td>3</td>
<td>25%</td>
<td>38%</td>
</tr>
<tr>
<td>BRAMs</td>
<td>32</td>
<td>16</td>
<td>0%</td>
<td>50%</td>
</tr>
<tr>
<td>DCM-ADVs</td>
<td>4</td>
<td>2</td>
<td>25%</td>
<td>50%</td>
</tr>
</tbody>
</table>

Table 4.7: The FPGA usage comparison of hardware-based Zigbee node and MicroBlaze CPU

RISC architecture. The CPU based implementation includes a 32Kbyte RAM for Zigbee protocol stack and application program. The synthesis results for MicroBlaze CPU and comparison to that of the hardware-based Zigbee node are shown in Table 4.7.

As illustrated in Table 4.7, the Zigbee implementation uses 52% of the total FPGA logical capacity and MicroBlaze implementation uses 13%. Since the FPGA logical capacity usage of a design is related to its circuit size, it is fair to say that the circuit size of hardware-based Zigbee implementation is about four times larger than that of the Mote style implementation with MicroBlaze CPU. The 4 input LUTs in the table is related to the size of combinational logic circuit. The size of combinational logic circuit in hardware-based implementation is about 8 times larger than that of MicroBlaze system. This is because the hardware-based design has many parallel data paths, among which computation units are distributed. Also, the flip flops usage in hardware-based system is higher, because there are more state machines in hardware-based system, and register files are used to store data instead of RAM.

Since just 52% of the FPGA resource is used, there is a room to implement a soft core CPU in our design to explore the system architecture in future. The CPU could be used either to replace some specific event handlers for reducing the circuit size or to implement complex applications.
Chapter 5

Conclusion and Future Work

This thesis focused on the prototyping a Zigbee sensor node using FPGAs. The Zigbee sensor node was implemented as a hardware-based system, in order to avoid the overhead of operating system and interrupt handling. Therefore it is more efficient in processing event-driven tasks than widely-used Mote style embedded sensor nodes. The main features of the prototype are presented below.

1. Event-driven architecture: Eliminates unnecessary event-processing overhead.

2. Hardware acceleration: The system consists of hardware units designed for specific tasks to improve performance.

3. Parallel Distributed Computation: Allows system to respond application’s needs faster.

Two Zigbee sensor nodes were implemented and tested for functionality verification and system performance. The test results show that the two sensor nodes are able to function as a Zigbee coordinator or a Zigbee router to form a Zigbee network and communicate with each other. The communication within 0 to 30 meters in the indoor environment and 0 to 65 meters in the outdoor environment is good, where the packet error rate is below 2%. Communication within 30 to 55
meters in the indoor environment and 65 to 85 meters in the outdoor environment is not that good, where packet error rate range from 5% to 20%. Although the quality of communication is not ideal in this range, the Zigbee system still functions. Beyond 55 meters in the indoor environment and 85 meters in the outdoor environment, Zigbee network is hard to establish.

The packet error induced by inferior connection between development boards was also quantitatively analyzed. It is shown that in high signal to noise environment, the packet error rate induced by connection cables is 1%.

The execution cycle count and system throughput are two parameters that we choose to evaluate the performance of the hardware-based event-driven Zigbee node. These two parameters are able to quantify the system efficiency in executing specific tasks. The test results show that our system provides significant improvement compared to the general processor solution. The implemented system is about 11 times faster than Mica2, a typical general CPU-based embedded sensor system, in packet transmission task, and about 3~4 times faster in packet reception and packet relay. The system is capable of achieving 250 Kbit/s throughput at 300 KHz system clock.

The performance gains in the sensor system are obtained by hardware acceleration and parallel computation features in our design. Based on our synthesis results, we estimate that the circuit size of our hardware-based system is about four time larger than a CPU-based embedded system. For our FPGA implementation platform, this trade-off is acceptable since only 52% of FPGA capacity is used in the implementation. For size constrained implementation, our design allows combining of several specific event handlers into a single general purpose event handler to reduce circuit size.

Future work that could follow this thesis is exploration of the sensor node design space and testing of different applications using the prototype nodes. The FPGA platform is a flexible medium for design space exploration and application development. To provide even more flexibility, an FPGA-based soft-core CPU could be integrated inside the implemented system, so that new applications could be developed in software or hardware. Such a system could be used to explore the trade-off between flexibility and performance.
References


REFERENCES


[34] www.xilinx.com Xilinx.


AppendixVHDL Source Code
Appendix A

VHDL Source Code

A.1 Zigbee Package

library ieee;
use ieee.std.logic.1164.all;

package zigbee.Constant.pkg is

-- CC2420 Command Register address

---------------------------
Constant SNOP_c : Std_logic_vector(7 downto 0) := "00000000";  -- 00
Constant SXOSCON_c : Std_logic_vector(7 downto 0) := "00000010";  -- 02
Constant STXCAL_c : Std_logic_vector(7 downto 0) := "00000011";  -- 03
Constant SRXON_c : Std_logic_vector(7 downto 0) := "00000001";  -- 01
### A. VHDL SOURCE CODE

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>STXON.C</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00000100&quot; ;</code></td>
</tr>
<tr>
<td><code>STXONCCA-c</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00000101&quot; ;</code></td>
</tr>
<tr>
<td><code>SRFOFF.C</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00000110&quot; ;</code></td>
</tr>
<tr>
<td><code>SXOSCOFFx</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00000111&quot; ;</code></td>
</tr>
<tr>
<td><code>SFLUSHRX.C</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00001000&quot; ;</code></td>
</tr>
<tr>
<td><code>SFLUSHTX-c</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00001001&quot; ;</code></td>
</tr>
<tr>
<td><code>SACKx</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00001010&quot; ;</code></td>
</tr>
<tr>
<td><code>SACKPENDx</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00001011&quot; ;</code></td>
</tr>
<tr>
<td><code>SRXDEC-C</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00001100&quot; ;</code></td>
</tr>
<tr>
<td><code>STXENC-C</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00001101&quot; ;</code></td>
</tr>
<tr>
<td><code>SAES.c</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00001110&quot; ;</code></td>
</tr>
<tr>
<td><code>SYNCWORDx</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00010100&quot; ;</code></td>
</tr>
<tr>
<td><code>TXCTRL-C</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00010101&quot; ;</code></td>
</tr>
<tr>
<td><code>RXCTRL0_c</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00010110&quot; ;</code></td>
</tr>
<tr>
<td><code>RXCTRLl.c</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00010111&quot; ;</code></td>
</tr>
<tr>
<td><code>FSCTRL.c</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00011000&quot; ;</code></td>
</tr>
<tr>
<td><code>SECCTRLO-c</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00011001&quot; ;</code></td>
</tr>
<tr>
<td><code>SECCTRLl.c</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00011010&quot; ;</code></td>
</tr>
<tr>
<td><code>BATTMONx</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00011011&quot; ;</code></td>
</tr>
<tr>
<td><code>IOCFG0-C</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00011100&quot; ;</code></td>
</tr>
<tr>
<td><code>IOCFG1.c</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00011101&quot; ;</code></td>
</tr>
<tr>
<td><code>MANFIDL.c</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00011110&quot; ;</code></td>
</tr>
<tr>
<td><code>MANFIDH.C</code></td>
<td><code>STD_LOGIC_VECTOR (7 downto 0) := &quot;00011111&quot; ;</code></td>
</tr>
</tbody>
</table>

---

### CC2420 Configuration Register Address

<table>
<thead>
<tr>
<th>Register Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&quot;04&quot;</code></td>
<td>Main Control Register</td>
</tr>
<tr>
<td><code>&quot;05&quot;</code></td>
<td>Modem Control Register 0</td>
</tr>
<tr>
<td><code>&quot;06&quot;</code></td>
<td>Modem Control Register 1</td>
</tr>
<tr>
<td><code>&quot;07&quot;</code></td>
<td>RSSI and CCA Status and Control register</td>
</tr>
<tr>
<td><code>&quot;0C&quot;</code></td>
<td>Synchronization word control register</td>
</tr>
<tr>
<td><code>&quot;15&quot;</code></td>
<td>Transmit Control Register</td>
</tr>
<tr>
<td><code>&quot;16&quot;</code></td>
<td>Receive Control Register 0</td>
</tr>
<tr>
<td><code>&quot;17&quot;</code></td>
<td>Receive Control Register 1</td>
</tr>
<tr>
<td><code>&quot;1B&quot;</code></td>
<td>Frequency Synthesizer Control/Status Register</td>
</tr>
<tr>
<td><code>&quot;1A&quot;</code></td>
<td>Security Control Register 0</td>
</tr>
<tr>
<td><code>&quot;1B&quot;</code></td>
<td>Battery Monitor Control and Status Register</td>
</tr>
<tr>
<td><code>&quot;1C&quot;</code></td>
<td>Input/Output Control Register 0</td>
</tr>
<tr>
<td><code>&quot;1D&quot;</code></td>
<td>Input/Output Control Register 1</td>
</tr>
<tr>
<td><code>&quot;1E&quot;</code></td>
<td>Manufacturer ID, Low 16 bits</td>
</tr>
<tr>
<td><code>&quot;1F&quot;</code></td>
<td>Manufacturer ID, High 16 bits</td>
</tr>
</tbody>
</table>
A. VHDL SOURCE CODE

Constant FSMTCx : std_logic_vector ( 7 downto 0 ) := "00100000" ; — x "20" ;
Finite State Machine Time Constants
Constant MANANDx : std_logic_vector ( 7 downto 0 ) := "00100001" ; — x "21" ;
Manual signal AND override register
Constant MANORx : std_logic_vector ( 7 downto 0 ) := "00100010" ; — x "22" ;
Manual signal OR override register
Constant AGCCTRL-C : std_logic_vector ( 7 downto 0 ) := "00100011" ; — x "23" ; AGC Control Register
Constant AGCTST0.c : std_logic_vector ( 7 downto 0 ) := "00100100" ; — x "24" ; AGC Test Register 0
Constant AGCTST1.c : std_logic_vector ( 7 downto 0 ) := "00100101" ; — x "25" ; AGC Test Register 1
Constant AGCTST2.c : std_logic_vector ( 7 downto 0 ) := "00100110" ; — x "26" ; AGC Test Register 2
Constant FSTST0.c : std_logic_vector ( 7 downto 0 ) := "00100111" ; — x "27" ; Frequency Synthesizer Test Register 0
Constant FSTST1.c : std_logic_vector ( 7 downto 0 ) := "00101000" ; — x "28" ; Frequency Synthesizer Test Register 1
Constant FSTST2.c : std_logic_vector ( 7 downto 0 ) := "00101001" ; — x "29" ; Frequency Synthesizer Test Register 2
Constant FSTST3.c : std_logic_vector ( 7 downto 0 ) := "00101010" ; — x "2A" ; Frequency Synthesizer Test Register 3
Constant RXBPFTST.c : std_logic_vector ( 7 downto 0 ) := "00101011" ; — x "2B" ; Receiver Bandpass Filter Test Register
Constant FSMSTATEx : std_logic_vector ( 7 downto 0 ) := "00101100" ; — x "2C" ; Finite State Machine State Status Register
Constant ADCTST.x : std_logic_vector ( 7 downto 0 ) := "00101101" ; — x "2D" ; ADC Test Register
Constant DACTST.x : std_logic_vector ( 7 downto 0 ) := "00101110" ; — x "2E" ; DAC Test Register
Constant TOPTST.x : std_logic_vector ( 7 downto 0 ) := "00101111" ; — x "2F" ; Top Level Test Register
-----------------------------------------------
--- read registers
-----------------------------------------------
Constant Read.MAIN.c : std_logic_vector ( 7 downto 0 ) := "01010000" ; — x "10" ; Main Control Register
Constant Read.MDMCTRLO.c : std_logic_vector ( 7 downto 0 ) := "01010001" ; — x "11" ; Modem Control Register 0
Constant Read.MDMCTRL1.c : std_logic_vector ( 7 downto 0 ) := "01010010" ; — x "12" ; Modem Control Register 1
Constant Read.RSSI.c : std_logic_vector ( 7 downto 0 ) := "01010100" ; — x "14" ; RSSI and CCA Status and Control register
Constant Read.SYNCTST.c : std_logic_vector ( 7 downto 0 ) := "01010101" ; — x "15" ; Synchronisation word control register
Constant Read.TXCTRL.c : std_logic_vector ( 7 downto 0 ) := "01010110" ; — x "16" ; Transmit Control Register
Constant Read.RXCTRL0.c : std_logic_vector ( 7 downto 0 ) := "01010111" ; — x "17" ; Receive Control Register 0
Constant Read.RXCTRL1.c : std_logic_vector ( 7 downto 0 ) := "01011000" ; — x "18" ; Frequency Synthesizer Ctrl&Status Register

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
A. VHDL SOURCE CODE

Constant Read_SECCTRLO_c : std_logic_vector (7 downto 0) := "01011001";
- z"19"; Security Control Register 0
Constant Read_SECCTRLLc : std_logic_vector (7 downto 0) := "01011010";
- z"1A"; Security Control Register 1
Constant Read_BATTMONc : std_logic_vector (7 downto 0) := "01011111";
- z"2F"; Battery Monitor Control and Status Register
Constant Read_JOCFG0,c : std_logic_vector (7 downto 0) := "01011100";
- z"1C"; Input/Output Control Register 0
Constant Read_JOCFG1,c : std_logic_vector (7 downto 0) := "01011101";
- z"1D"; Input/Output Control Register 1
Constant Read_MANFIDLc : std_logic_vector (7 downto 0) := "01011110";
- z"21"; Manufacturer ID, Low 16 bits
Constant Read_MANFIDH.c : std_logic_vector (7 downto 0) := "01011111";
- z"22"; Manufacturer ID, High 16 bits
Constant Read_FSMTC.c : std_logic_vector (7 downto 0) := "01100000";
- z"20"; Finite State Machine Time Constants
Constant Read_MANAND.c : std_logic_vector (7 downto 0) := "01100001";
- z"23"; Manual signal AND override register
Constant Read_MANOR.c : std_logic_vector (7 downto 0) := "01100010";
- z"24"; Manual signal OR override register
Constant Read_AGCTST0_c : std_logic_vector (7 downto 0) := "01100100";
- z"24"; AGC Test Register 0
Constant Read_AGCTST1.c : std_logic_vector (7 downto 0) := "01100101";
- z"25"; AGC Test Register 1
Constant Read_AGCTST2_c : std_logic_vector (7 downto 0) := "01100110";
- z"26"; AGC Test Register 2
Constant Read_FSTST0.c : std_logic_vector (7 downto 0) := "01100111";
- z"27"; Frequency Synthesizer Test Register 0
Constant Read_FSTST1.c : std_logic_vector (7 downto 0) := "01101000";
- z"28"; Frequency Synthesizer Test Register 1
Constant Read_FSTST2.c : std_logic_vector (7 downto 0) := "01101001";
- z"29"; Frequency Synthesizer Test Register 2
Constant Read_FSTST3.c : std_logic_vector (7 downto 0) := "01101010";
- z"2A"; Frequency Synthesizer Test Register 3
Constant Read_RXBPFTST.c : std_logic_vector (7 downto 0) := "01101011";
- z"2B"; Receiver Bandpass Filter Test Register
Constant Read_FSMBSTATE.c : std_logic_vector (7 downto 0) := "01101100";
- z"2C"; Finite State Machine State Status Register
Constant Read_ADCTST.c : std_logic_vector (7 downto 0) := "01101101";
- z"2D"; ADC Test Register
Constant Read_DACST.c : std_logic_vector (7 downto 0) := "01101110";
- z"2E"; DAC Test Register
Constant Read_TOPTST.c : std_logic_vector (7 downto 0) := "01101111";
- z"2F"; Top Level Test Register

-----------------------------+-----------------------------
- z"3E"; Transmit FIFO Byte Register

Constant TXFIFO.c : std_logic_vector (7 downto 0) := "00111110"; - z"3E"; --

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
A. VHDL SOURCE CODE

Constant RXFIFO.c : std_logic_vector ( 7 downto 0) := "01111111" ; —x "7F" ; — Read Receiver FIFO Byte Register
Constant RXFIFO.add.c : std_logic_vector ( 7 downto 0) := "00111111" ; —x "3F" ; — Receiver FIFO Byte Register

--------------
→ CC2430 RAM Address +
--------------
Constant SHORTADR.MSB.MEMBank.Address.c: Std_logic_vector(1 downto 0) := "10" ; — RAM BANK address for 16-bit Short address,
Constant SHORTADR.MSB.MEM.Address.c: Std_logic_vector(6 downto 0) := "1101011" ; —x "69" RAM address for 16-bit Short address,
Constant SHORTADR.LSB.MEMBank.Address.c: Std_logic_vector(1 downto 0) := "10" ; —16-bit Short address, for address recognition.
Constant SHORTADR.LSB.MEM.Address.c: Std_logic_vector(6 downto 0) := "1101010" ; —x "6A" ; 16-bit Short address
Constant PANID.MSB.MEMBank.Address.c: Std_logic_vector(1 downto 0) := "10" ; — 16-bit PAN identifier, for address recognition.
Constant PANID.MSB.MEM.Address.c: Std_logic_vector(6 downto 0) := "1101001" ; —x "69" ; 16-bit PAN identifier.
Constant PANID.LSB.MEMBank.Address.c: Std_logic_vector(1 downto 0) := "10" ; — RAM BANK address for 16-bit Short address,
Constant PANID.LSB.MEM.Address.c: Std_logic_vector(6 downto 0) := "1101000" ; —x 68 ; 16-bit PAN identifier.
Constant IEEEaddr.MSB.MEMBank.Address.c: Std_logic_vector(1 downto 0) := "10" ; — RAM BANK address for 64-bit IEEE address
Constant IEEEaddr.MSB.MEM.Address.c: Std_logic_vector(6 downto 0) := "1100101" ; —x "65" RAM address for 16-bit Short address,
Constant IEEEaddr.2ndB.MEMBank.Address.c: Std_logic_vector(1 downto 0) := "10" ; — RAM BANK address for 64-bit IEEE address
Constant IEEEaddr.2ndB.MEM.Address.c: Std_logic_vector(6 downto 0) := "1100100" ; —x "64" RAM address for 16-bit Short address,
Constant IEEEaddr.3thB.MEMBank.Address.c: Std_logic_vector(1 downto 0) := "10" ; — RAM BANK address for 64-bit IEEE address
Constant IEEEaddr.3thB.MEM.Address.c: Std_logic_vector(6 downto 0) := "1100101" ; —x "65" RAM address for 16-bit Short address,
Constant IEEEaddr.4thB.MEMBank.Address.c: Std_logic_vector(1 downto 0) := "10" ; — RAM BANK address for 64-bit IEEE address
Constant IEEEaddr.4thB.MEM.Address.c: Std_logic_vector(6 downto 0) := "1100100" ; —x "64" RAM address for 16-bit Short address,
Constant IEEEaddr.5thB.MEMBank.Address.c: Std_logic_vector(1 downto 0) := "10" ; — RAM BANK address for 64-bit IEEE address
Constant IEEEaddr.5thB.MEM.Address.c: Std_logic_vector(6 downto 0) := "1100111" ; —x "63" RAM address for 16-bit Short address,
Constant IEEEaddr.6thB.MEMBank.Address.c: Std_logic_vector(1 downto 0) := "10" ; — RAM BANK address for 64-bit IEEE address
Constant IEEEaddr.6thB.MEM.Address.c: Std_logic_vector(6 downto 0) := "1100010" ; —x "62" RAM address for 16-bit Short address,
Constant IEEEaddr.7thB.MEMBank.Address.c: Std_logic_vector(1 downto 0) := "10" ; — RAM BANK address for 64-bit IEEE address
Constant IEEEaddr.7thB.MEM.Address.c: Std_logic_vector(6 downto 0) := "1100001" ; —x "61" RAM address for 16-bit Short address,

54

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
A. VHDL SOURCE CODE

Constant IEEEaddr.LSB.MEMBank.Address.c : Std.logic.vector(1 downto 0) := "10";
— RAM BAND address for 64-bit IEEE address
Constant IEEEaddr.LSB.MEM.Address.c : Std.logic.vector(6 downto 0) := "1100000";
— x^60 RAM address for 16-bit Short address,

— TRX unit Constant
constant MinPacketLength.c: natural := 5;
— Used for check if receive packet is correct length; shortest frame is Ack frame is 5;
constant MaxFIFOsize.c: natural := 125;
— the Max FIFO in CC2420, it also the MAX length of a frame;
constant Data_len.g.c : natural := 8;
— the size of data path
constant Count_for.IMS.c: natural := 4800; — set to 10 for simulation
constant Count_for.IMS.c: natural := system.Frequency/1000;

— from pars
Constant MAC.Assoc_req.c : Std.logic.vector(7 downto 0) := "00000001";
Constant MAC.Assoc.resp.c : Std.logic.vector(7 downto 0) := "00000010";
Constant MAC.Beacon.req.c : Std.logic.vector(7 downto 0) := "00000111";
Constant MAC.beacon.receive.c : Std.logic.vector(7 downto 0) := "000001010";
Constant MAC.ACK.receive.c : Std.logic.vector(7 downto 0) := "00000111";
Constant MAC.reset.c : Std.logic.vector(7 downto 0) := "00000111";
Constant MAC.join.network.c : Std.logic.vector(7 downto 0) := "0000011100";
Constant MAC.search.network.c : Std.logic.vector(7 downto 0) := "0000011101";
Constant MAC.send.NWKdata.c : Std.logic.vector(7 downto 0) := "0000011110";
Constant MAC.Start.NWK.c : Std.logic.vector(7 downto 0) := "000001111";

— from NWK
Constant MAC.Assoc_req.c : Std.logic.vector(7 downto 0) := "00000001";
Constant MAC.Assoc.resp.c : Std.logic.vector(7 downto 0) := "00000010";
Constant MAC.Beacon.req.c : Std.logic.vector(7 downto 0) := "00000111";
Constant MAC.ACK.receive.c : Std.logic.vector(7 downto 0) := "00000111";
Constant MAC.reset.c : Std.logic.vector(7 downto 0) := "00000111";
Constant MAC.join.network.c : Std.logic.vector(7 downto 0) := "0000011100";
Constant MAC.search.network.c : Std.logic.vector(7 downto 0) := "0000011101";
Constant MAC.send.NWKdata.c : Std.logic.vector(7 downto 0) := "0000011110";
Constant MAC.Start.NWK.c : Std.logic.vector(7 downto 0) := "000001111"; 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
A. VHDL Source Code

-- from Pars
Constant NWK.Route.req.c : Std.logic.vector(7 downto 0) := "00000001"; -- 301
Constant NWK.Route.reply.c : Std.logic.vector(7 downto 0) := "00000010"; -- 302
Constant NWK.Route.error.c : Std.logic.vector(7 downto 0) := "00000011" ; -- 303

-- from MAC
Constant NWK.Associate.Confirm.c : Std.logic.vector(7 downto 0) := "00000100"; -- 304
Constant NWK.SendData.Confirm.c : Std.logic.vector(7 downto 0) := "00000101"; -- 305
Constant NWK.beacon.receive.c : Std.logic.vector(7 downto 0) := "00000110" ; -- 306
Constant NWK.Start.Confirm.c : Std.logic.vector(7 downto 0) := "00000111" ; -- 307

const variable not in network constant 

Constant NWKinfo.not.in.netowrk.c : Std.logic.vector(7 downto 0) := "00000001" ; -- 301
Constant NWKinfo.join.network.success.c : Std.logic.vector(7 downto 0) := "00000010"; -- 302
Constant NWKinfo.invali.d.req.c : Std.logic.vector(7 downto 0) := "00000011" ; -- 303
Constant NWKinfo.no.nwk.available.c : Std.logic.vector(7 downto 0) := "00000100" ; -- 304
Constant NWKinfo.join.network.unsuccess.c : Std.logic.vector(7 downto 0) := "00000101" ; -- 305
Constant NWKinfo.Start.network.success.c : Std.logic.vector(7 downto 0) := "00000110" ; -- 306
Constant NWKinfo.new.network.found.c : Std.logic.vector(7 downto 0) := "00000111" ; -- 307
Constant NWKinfo.new.Neighbor.found.c : Std.logic.vector(7 downto 0) := "00001000" ; -- 308

--++ test.bench constant ++

constant period.c : time := 400 ns; -- 2.5 MHz frequency

--++ communication type constant ++

Constant idle : Std.logic.vector(1 downto 0) := "00" ; -- 300
Constant send1 : Std.logic.vector(1 downto 0) := "01" ; -- 300
Constant send2 : Std.logic.vector(1 downto 0) := "10" ; -- 300
Constant receive1 : Std.logic.vector(1 downto 0) := "01" ; -- 300
Constant receive2 : Std.logic.vector(1 downto 0) := "10" ; -- 300

56
A.2 Zigbee Top Module

library ieee;
library ZigbeeLib;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ZigbeeLib.zigbee.constant.pkg.all;
use ZigbeeLib.zigbee.type.pkg.all;

entity ZigbeeNode is
port (  
--> system interface  
   clock.48MHz: in std_logic;  
   reset : in std_logic;  

--> display interface  
   LCD1.out: out std_logic_vector(7 downto 0);  
   LCD2.out: out std_logic_vector(7 downto 0);  
   LED0.out: out std_logic;  
   LED1.out: out std_logic;
end;
A. VHDL SOURCE CODE

LED2_out: out std_logic;
LED3_out: out std_logic;
LED4_out: out std_logic;
LED5_out: out std_logic;
LED6_out: out std_logic;
LED7_out: out std_logic;

--** CC2420 interface
SO: out std_logic;
SI: in std_logic;
SCLK: out std_logic;
CSn: out std_logic;
FIFO: in std_logic;
FIFOP: in std_logic;
CCA: in std_logic;
SFD: in std_logic;

Vreg.EN: out std_logic;
Resetn_out: out std_logic
);

end ZigbeeNode;

architecture structural of ZigbeeNode is

COMPONENT ClockModule
PORT(
  CLKN_IN : IN std_logic;
  RST.IN : IN std_logic;
  CLKDV.OUT: OUT std_logic;
  CLKINJBUF.OUT: OUT std_logic;
  CLKO.OUT : OUT std_logic;
  LOCKED.OUT : OUT std_logic
);
END COMPONENT;

----------------------------------------------------------------------------------
--** Define components **
----------------------------------------------------------------------------------

component TX.RX.Control.Unit
port {
  clock: in std_logic;
  resetn : in std_logic;

  Data_to_parsing_out: out std_logic_vector ( 7 downto 0);
  communication_control_to_parsing_out: out
  communication_control.Master_type;
  communication_control_to_parsing_in: in communication_control.slave_type
};
Data from Frame in: in std_logic_vector (7 downto 0);
communication control from Frame in: in
communication control master type;
communication control from Frame out: out
communication control slave type;

Data from MAC in: in std_logic_vector (7 downto 0);
Communication control from MAC in: in Communication Control master type;
communication control from MAC out: out communication control slave type;

Data to MAC out: out std_logic_vector (7 downto 0);
Communication Control to MAC out: out Communication Control master type;
Communication control to MAC in: in communication control slave type;

SO: out std_logic;
SI: in std_logic;
SCLK: out std_logic;
CSn: out std_logic;

FIFO: in std_logic;
FIFOP: in std_logic;
CCA: in std_logic;
SFD: in std_logic;

---* test ports definition
LED0 Receive RXFIFO Begin: out std_logic;
LED1 Receive RXFIFO Complete: out std_logic;
LED2 Access TXFIFO: out std_logic;
LED3 Send Command Complete: out std_logic;
LED4 CCA Clear: out std_logic;
LED5 Done Send Packet: out std_logic;
LED6 Sent Command Begin: out std_logic;
LED7 Send Command Complete: out std_logic
---* end of test port definition

end component;

component Frame Unit
port ( clock : in std_logic;
reseta : in std_logic;
From Pars : in std_logic_vector (7 downto 0);
From Pars Ctrl MS in: in Communication Control master type;
From Pars Ctrl SI out: out communication control slave type;
From MAC : in std_logic_vector (7 downto 0);
From MAC Ctrl MS in: in Communication Control master type;
From MAC Ctrl SI out: out communication control slave type;

59

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
ToTRX.out: Out std_logic_vector ( 7 downto 0);
ToTRX.Ctrl_MS.out: out Communication.Control.master_type;
ToTRX.Ctrl_SL.in: in communication.control.slave_type;

FromNWK.in: In std_logic_vector ( 7 downto 0);
FromNWK.Ctrl_MS.in: in Communication.Control.master_type;
FromNWK.Ctrl_SL.out: out communication.control.slave_type;

ToNWK.out: Out std_logic_vector ( 7 downto 0);
ToNWK.Ctrl_MS.out: out Communication.Control.master_type;
ToNWK.Ctrl_SL.in: in communication.control.slave_type;

NWK.PANID_H.in: in std_logic_vector ( 7 downto 0);
NWK.PANID_L.in: in std_logic_vector ( 7 downto 0);
NWK.shortAddr_H.in: in std_logic_vector ( 7 downto 0);
NWK.shortAddr_L.in: in std_logic_vector ( 7 downto 0);

--*-- test ports definition
LED0.Pars.Receive: out std_logic;
LED1.MAC.Receive: out std_logic;
LED2.receive.Complete: out std_logic;
LED4.RoutingInfo.Receive: out std_logic;
LED5.Routing.Finish: out std_logic;
LED6.Frame_Begin.send: out std_logic;
LED7_FrameSent_Complete: out std_logic

--*-- end of test ports definition
}

end component; -- Frame_unit;

component parsing_Unit
port (
  clock : in std_logic;
  Resetn : in std_logic;

  FromTRX.Data.in: in std_logic_vector (7 downto 0);
  FromTRX.Control_MS.in: in communication.control.master_type;
  FromTRX.Control_SL.out: out communication.control.slave_type;

  toMAC.Data.out: out std_logic_vector (7 downto 0);
  toMAC.Control_MS.out: out communication.control.master_type;
  toMAC.Control_SL.in: in communication.control.slave_type;

  toNWK.Data.out: out std_logic_vector (7 downto 0);
  toNWK.Control_MS.out: out communication.control.master_type;
  toNWK.Control_SL.in: in communication.control.slave_type;

  toAPS.Data.out: out std_logic_vector (7 downto 0);
  toAPS.Control_MS.out: out communication.control.master_type;
  toAPS.Control_SL.in: in communication.control.slave_type;

  toFrame.Data.out: out std_logic_vector (7 downto 0);
);
A. VHDL SOURCE CODE

toFrame.control_MS.out: out communication.control_master.type;
toFrame.control SL.in: in communication.control_slave.type;

NWK.shortAddr.H.in: in std.logic.vector (7 downto 0);
NWK.shortAddr.L_in: in std.logic.vector (7 downto 0);

---* test ports definition
LED0.Receive.TRX: out std.logic;
LED1.Pars.Complete: out std.logic;
LED2.Beacon_read: out std.logic;
LED3.APSData.Receive: out std.logic;
LED4.MACCommand.Receive: out std.logic;
LED5.Parsing.Done: out std.logic;
LED6.Parsing.result_sent: out std.logic;
LED7.Parsing.all.done: out std_logic

---* end of test ports definition

end component; -- parsing_unit;

component ZigbeeMAC
generic (zigbeeNodeID: integer:=1);
port (
  clock : in std_logic;
  resetn : in std_logic;

  FromTRX.in: in std.logic.vector (7 downto 0);
  FromTRX.Ctrl.MS.in: in Communication.Control.master.type;
  FromTRX.Ctrl.SL.out: out communication.control.slave.type;

  ToTRX.out: Out std_logic.vector (7 downto 0);
  ToTRX.Ctrl.MS.out: out Communication.Control.master.type;
  ToTRX.Ctrl.SL.in: in communication.control.slave.type;

  FromNWK.in: in std_logic.vector (7 downto 0);
  FromNWK.Ctrl.MS.in: in Communication.Control.master.type;
  FromNWK.Ctrl.SL.out: out communication.control.slave.type;

  ToNWK.out: Out std_logic.vector (7 downto 0);
  ToNWK.Ctrl.MS.out: out Communication.Control.master.type;
  ToNWK.Ctrl.SL.in: in communication.control.slave.type;

  ToFrame.out: Out std_logic.vector (7 downto 0);
  ToFrame.Ctrl.MS.out: out Communication.Control.master.type;
  ToFrame.Ctrl.SL.in: in communication.control.slave.type;

  FromPars.in: in std_logic.vector (7 downto 0);
  FromPars.Ctrl.MS.in: in Communication.Control.master.type;
  FromPars.Ctrl.SL.out: out communication.control.slave.type;

  NWK.PANID_H.out: out std_logic.vector (7 downto 0);
  NWK.PANID_L.out: out std_logic.vector (7 downto 0);
  NWK.shortAddr.H.out: out std_logic.vector (7 downto 0);

end component;
A. VHDL SOURCE CODE

NWK_shortAddr_L.out: out std_logic_vector (7 downto 0);
NWK_Parent_shortaddr_H.out: out std_logic_vector (7 downto 0);
NWK_Parent_shortaddr_L.out: out std_logic_vector (7 downto 0);
NWKDepth.out: out std_logic_vector (1 downto 0);

---**
test ports definition

LED0_CC2420_OSC_Stable: out std_logic;
LED1_initial_CC2420_Complete: out std_logic;
LED2_Receive_ParsInfo_complete: out std_logic;
LED3_Receive_NWKInfo_complete: out std_logic;
LED4_Associate_Req_Process_Complete: out std_logic;
LED5_Associate RESP_Process_Complete: out std_logic;
LED6_ToCC2420_Complete: out std_logic;
LED7_ToFrame_Complete: out std_logic;
---**
end of test ports definition

MAC_state.out: out std_logic;
MS_Flag.out: out std_logic;
end component; — ZigbeeMAC;

component ZigbeeNWK
port
(
    clock : in std_logic;
    resetn : in std_logic;
    FromMAC_in: in std_logic_vector (7 downto 0);
    FromMAC_Ctrl_MS_in: in Communication_Control_master_type;
    FromMAC_Ctrl_SL_out: out communication_control_slave_type;
    ToMAC_out: Out std_logic_vector (7 downto 0);
    ToMAC_Ctrl_MS_out: out Communication_Control_master_type;
    ToMAC_Ctrl_SL_in: in communication_control_slave_type;
    FromAPS_in: in std_logic_vector (7 downto 0);
    FromAPS_Ctrl_MS_in: in Communication_Control_master_type;
    FromAPS_Ctrl_SL_out: out communication_control_slave_type;
    ToFrame_out: Out std_logic_vector (7 downto 0);
    ToFrame_Ctrl_MS_out: out Communication_Control_master_type;
    ToFrame_Ctrl_SL_in: in communication_control_slave_type;
    FromFrame_in: in std_logic_vector (7 downto 0);
    FromFrame_Ctrl_MS_in: in Communication_Control_master_type;
    FromFrame_Ctrl_SL_out: out communication_control_slave_type;
    FromPars_in: in std_logic_vector (7 downto 0);
    FromPars_Ctrl_MS_in: in Communication_Control_master_type;
    FromPars_Ctrl_SL_out: out communication_control_slave_type;
    NWK_PANID_H_in: in std_logic_vector (7 downto 0);
)
end component; — ZigbeeMAC;
A. VHDL SOURCE CODE

NWK.PANID.L.in: in std_logic_vector (7 downto 0);
NWK.shortAddr.H.in: in std_logic_vector (7 downto 0);
NWK.shortAddr.L.in: in std_logic_vector (7 downto 0);
NWK.Parent.shortaddr.H.in: in std_logic_vector (7 downto 0);
NWK.Parent.shortaddr.L.in: in std_logic_vector (7 downto 0);
NWKDepth.in: in std_logic_vector (1 downto 0);
MAC.state.in: in std_logic;
MS.Flag.in: in std_logic;

LCD1.out: out std_logic_vector(7 downto 0);
LCD2.out: out std_logic_vector(7 downto 0);

---** test ports definition
LED0.Search.Network: out std_logic;
LED1.Network.found: out std_logic;
LED2.Network.join: out std_logic;
LED3.Work.as.coordinator: out std_logic;
LED4.RoutingInfo.Sent: out std_logic;
LED5.NWK.data.receive: out std_logic;
LED6.NWK.data.sent: out std_logic;
LED7.TimePacke.sent: out std_logic
---** end of test ports definition

end component; — ZigbeeMAC;

---******************************************************************************
---** Define connection wires **
---******************************************************************************
signal FromMAC.ToFrame.Data: std_logic_vector (7 downto 0);
signal FromMAC.ToFrame.Ctrl.MS: Communication.Control.master.type;
signal FromMAC.ToFrame.Ctrl.Sl: communication.control.slave.type;

---******************************************************************************
---** Define connection wires **
---******************************************************************************
signal FromMAC.ToTRX.Data: std_logic_vector (7 downto 0);
signal FromMAC.ToTRX.Ctrl.MS: Communication.Control.master.type;
signal FromMAC.ToTRX.Ctrl.Sl: communication.control.slave.type;

---******************************************************************************
---** Define connection wires **
---******************************************************************************
signal FromMAC.ToNWK.Data: std_logic_vector (7 downto 0);
signal FromMAC.ToNWK.Ctrl.MS: Communication.Control.master.type;
signal FromMAC.ToNWK.Ctrl.Sl: communication.control.slave.type;

---******************************************************************************
---** Define connection wires **
---******************************************************************************
signal FromFrame.ToTRX.Data: std_logic_vector (7 downto 0);
signal FromFrame.ToTRX.Ctrl.MS: Communication.Control.master.type;
signal FromFrame.ToTRX.Ctrl.Sl: communication.control.slave.type;

---******************************************************************************
---** Define connection wires **
---******************************************************************************
signal FromFrame.ToNWK.Data: std_logic_vector (7 downto 0);
signal FromFrame.ToNWK.Ctrl.MS: Communication.Control.master.type;
signal FromFrame.ToNWK.Ctrl.Sl: communication.control.slave.type;

---******************************************************************************
---** Define connection wires **
---******************************************************************************
signal FromNWK.ToFrame.Data: std_logic_vector (7 downto 0);
signal FromNWK.ToFrame.Ctrl.MS: Communication.Control.master.type;
signal FromNWK.ToFrame.Ctrl.Sl: communication.control.slave.type;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
A. VHDL SOURCE CODE

```vhdl
signal FromNWK.ToMAC.Data: std_logic_vector (7 downto 0);
signal FromNWK.ToMAC.CtrlMS: Communication_Control_master_type;
signal FromNWK.ToMAC.CtrlSL: communication_control_slave_type;
signal FromTRX.ToPars.Data: std_logic_vector (7 downto 0);
signal FromTRX.ToPars.Ctrl.MS: communication.control.master.type;
signal FromTRX.ToPars.Ctrl.SL: communication.control.slave.type;
signal FromTRX.ToMAC.Data: std_logic_vector (7 downto 0);
signal FromTRX.ToMAC.CtrlMS: Communication_Control_master_type;
signal FromTRX.ToMAC.CtrlSL: communication.control.slave.type;
signal FromPars.toAPS.Data: std_logic_vector (7 downto 0);
signal FromPars.toAPS.Ctrl.MS: communication.control.master.type;
signal FromPars.toAPS.Ctrl.SL: communication.control.slave.type;
signal FromPars.ToFrame.Data: std_logic_vector (7 downto 0);
signal FromPars.ToFrame.Ctrl.MS: Communication_Control_master_type;
signal FromPars.ToFrame.Ctrl.SL: communication.control.slave.type;
signal FromPars.toMAC.Data: std_logic_vector (7 downto 0);
signal FromPars.toMAC.CtrlMS: communication.control.master.type;
signal FromPars.toMAC.CtrlSL: communication.control.slave.type;
signal FromPars.toNWK.Data: std_logic_vector (7 downto 0);
signal FromPars.toNWK.CTRL.MS: communication.control.master.type;
signal FromPars.toNWK.CTRL.SL: communication.control.slave.type;
signal NWK.PANID.H: std_logic_vector (7 downto 0);
signal NWK.PANID.L: std_logic_vector (7 downto 0);
signal NWK.shortAddr.H: std_logic_vector (7 downto 0);
signal NWK.shortAddr.L: std_logic_vector (7 downto 0);
signal NWK.Parent.shortAddr.H: std_logic_vector (7 downto 0);
signal NWK.Parent.shortAddr.L: std_logic_vector (7 downto 0);
signal NWKDepth: std_logic_vector (1 downto 0);
signal MS.Flag: std_logic;
signal MAC.state: std_logic;
```

---

test port for Zigbee NWK

```vhdl
signal LED0: std_logic;
signal LED1: std_logic;
signal LED2: std_logic;
signal LED3: std_logic;
signal LED4: std_logic;
signal LED5: std_logic;
signal LED6: std_logic;
signal LED7: std_logic;
signal clock: std_logic;
signal resetn: std_logic;
```

---

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
A. VHDL SOURCE CODE

---************ Beginning of the structure style program  ********
--- The test ports could be connected to LED or just leave  ****
--- it open, based on the need of debugging  ********

begin

Resetn<= not reset;
Vreg.EN<= '1';
Resetn_out<= Resetn;
LED0_out<= LED0;
LED1_out<= LED1;
LED2_out<= LED2;
LED3_out<= LED3;
LED4_out<= LED4;
LED5_out<= LED5;
LED6_out<= LED6;
LED7_out<= LED7;

UI: Frame_unit
 port map (  
clock => clock,
reset => Resetn,

FromPars.in=> FromPars.ToFrame.Data,
FromPars.Ctrl.MS.in=>FromPars.ToFrame.Ctrl.MS,
FromPars.Ctrl.SL.out=>FromPars.ToFrame.Ctrl.SL,

FromMAC.in=> FromMAC.ToFrame.Data,
FromMAC.Ctrl.MS.in=>FromMAC.ToFrame.Ctrl.MS,
FromMAC.Ctrl.SL.out=>FromMAC.ToFrame.Ctrl.SL,

ToTRX.out=>FromFrame.ToTRX.Data,
ToTRX.Ctrl.MS.out=>FromFrame.ToTRX.Ctrl.MS,
ToTRX.Ctrl.SL.in=>FromFrame.ToTRX.Ctrl.SL,

FromNWK.in=>FromNWK.ToFrame.Data,
FromNWK.Ctrl.MS.in=>FromNWK.ToFrame.Ctrl.MS,
FromNWK.Ctrl.SL.out=>FromNWK.ToFrame.Ctrl.SL,

ToNWK.out=>FromFrame.ToNWK.Data,
ToNWK.Ctrl.MS.out=>FromFrame.ToNWK.Ctrl.MS,
ToNWK.Ctrl.SL.in=>FromFrame.ToNWK.Ctrl.SL,

NWK_PANID.H.out=>NWK.PANID.H,
NWK_PANID.L.out=>NWK.PANID.L,
NWK.shortAddr.H.in=>NWK.shortAddr.H,
NWK.shortAddr.L.in=>NWK.shortAddr.L,

---** test port  ------
LED0.Pars.Receive=>open,
LED1.MAC.Receive=>open,
A. VHDL SOURCE CODE

LED2.receive.Complete=>open,
LED3.Send.Routing.Req=>open,
LED4.RoutingInfo.Receive=>open,
LED5.Routing.Finish=>open,
LED6.Frame.Begin.send=>open,
LED7.FrameSent.Complete=>open
);

U2: parsing.unit
port map (  
clock=> clock ,
Resetn=> Resetn ,
  
FromTRX_Data.in=>FromTRX.ToPars_Data ,
FromTRX.control.MS.in=>FromTRX.ToPars.Ctrl_MS ,
FromTRX.control.SL.out=> FromTRX.ToPars.Ctrl.SL ,

toMAC.Data.out=>Frompars.toMAC.Data ,
toMAC.control.MS.out=>Frompars.toMAC.Ctrl_MS ,
toMAC.control.SL.in=> Frompars.toMAC.Ctrl.SL ,

toNWK.Data.out=>Frompars.toNWK.Data ,
toNWK.control.MS.out=>Frompars.toNWK.Ctrl_MS ,
toNWK.control.SL.in=> Frompars.toNWK.Ctrl.SL ,

toAPS.Data.out=>Frompars.toAPS.Data ,
toAPS.control.MS.out=>Frompars.toAPS.Ctrl_MS ,
toAPS.control.SL.in=> Frompars.toAPS.Ctrl.SL ,

toFrame.Data.out=>Frompars.toFrame.Data ,
toFrame.control.MS.out=>Frompars.toFrame.Ctrl_MS ,
toFrame.control.SL.in=> Frompars.toFrame.Ctrl.SL ,

NWK.shortAddr.H_in=>NWK.shortAddr.H ,
NWK.shortAddr.L_in=>NWK.shortAddr.L ,

--- ** test port ---------
LED0.Receive.TRX=>open,
LED1.Pars.Complete=>open,
LED2.Beacon.read=>open,
LED3.APSData.Receive=>open,
LED4.MACCommand.Receive=>open,
LED5.Parsing.Done=>open,
LED6.Parsing.result.sent=>open,
LED7.Parsing.all.done=>open
);

U3: TX.RX.Control.Unit
port map (  
clock => CLocK ,
resetn => Resetn ,

---
A. VHDL SOURCE CODE

---

**test port**

- LED0.Receive.RXFIFO.Begin=>open
- LED1.Receive.RXFIFO.Complete=>open
- LED2.Access.TXFIFO=>open
- LED4.CCA.Clear=>open
- LED5.Done.Send.Packet=>open
- LED6.sent.Command.begin=>open
- LED7.Send.Command.Complete=>open

U4: ZigbeeMAC

**generic map** (ZigbeeNodeID=>1)

**port map**

- clock=> clock
- Resctn=> Resctn

- FromTRX.in=>FromTRX.ToMAC.Data
- FromTRX.Ctl.MS.in=>FromTRX.ToMAC.Ctl.MS
- FromTRX.Ctl.SL.out=>FromTRX.ToMAC.Ctl.SL

- ToTRX.out=>FromMAC.ToTRX.Data
- ToTRX.Ctl.MS.out=>FromMAC.ToTRX.Ctl.MS
- ToTRX.Ctl.SL.in=>FromMAC.ToTRX.Ctl.SL

- FromNWK_in=>FromNWK.ToMAC.Data
- FromNWK.Ctl.MS.in=>FromNWK.ToMAC.Ctl.MS

---
FromNWK_Ctrl_Sl_out=>FromNWK_ToMAC_Ctrl_Sl,
ToNWK_out=>FromMAC.ToNWK_Data,
ToNWK_Ctrl_MS.out=>FromMAC.ToNWK_CtrlMS,
ToNWK_Ctrl_Sl.in=>FromMAC.ToNWK_Ctrl_Sl,
ToFrame.out=>FromMAC.ToFrame_Data,
ToFrame_Ctrl_MS_out=>FromMAC.ToFrame_CtrlMS,
ToFrame_Ctrl_Sl.in=>FromMAC.ToFrame_Ctrl_Sl,
FromPars_in=>FromPars.toMAC.Data,
FromPars_Ctrl_MS_in=>FromPars.toMAC_CtrlMS,
FromPars_Ctrl_Sl_out=>FromPars_toMAC_Ctrl_Sl,
NWK_PANID_H_out=>NWK_PANID_H,
NWK_PANID_L_out=>NWK_PANID_L,
NWK short Addr_H_out=>NWK short Addr_H,
NWK short Addr_L_out=>NWK short Addr_L,
NWK_Parent_shortaddr_H_out=>NWK_Parent_shortaddr_H,
NWK_Parent_shortaddr_L_out=>NWK_Parent_shortaddr_L,
NWKDepth.out=>NWKDepth,

--- * * test port -----
LED0_CC2420_OSC_Stable=>LED0,
LED1_initial_CC2420_Complete=>LED1,
LED2_Receive_ParsInfo_complete=>LED2,
LED3_Receive_NWKInfo_complete=>LED3,
LED4_Associate_Req_Process_Complete=>LED4,
LED5_Associate_Resp_Process_Complete=>LED5,
LED6_ToCC2420_Complete=>LED6,
LED7_ToFrame_Complete=>LED7,

--- * * end test port ---
MAC_state.out=>MAC_state,
MS_Flag.out=>MS_Flag
);

U5: ZigbeeNWK
port map (    clock=> clock,
Resetn=> Resetn,
FromMAC_in=>FromMAC_toNWK_Data,
FromMAC_Ctrl_MS_in=>FromMAC_toNWK_CtrlMS,
FromMAC_Ctrl_Sl.out=>FromMAC_toNWK_Ctrl_Sl,
ToMAC_out=>FromNWK_toMAC_Data,
ToMAC_Ctrl_MS.out=>FromNWK_toMAC_CtrlMS,
ToMAC_Ctrl_Sl.in=>FromNWK_toMAC_Ctrl_Sl,
FromAPS_in=>FromPars_toAPS_Data,
FromAPS_Ctrl_MS.in=>FromPars_toAPS_Ctrl_MS,
FromAPS_Ctrl_Sl.out=>FromPars_toAPS_Ctrl_Sl,
A. VHDL SOURCE CODE

```vhdl
ToFrame_out => FromNWK.ToFrame_Data,
ToFramc_Ctrl_MS_out => FromNWK_ToFrame_Ctrl_MS,
ToFramc_Ctrl Sl.in => FromNWK.ToFrame_Ctrl_Sl,

FromFrame.in => FromFrame.ToNWK_Data,
FromFrame_Ctrl_MS.in => FromFrame.ToNWK_Ctrl_MS,
FromFrame_Ctrl Sl.out => FromFrame.ToNWK_Ctrl_Sl,

FromPars.in => FromPars.toNWK_Data,
FromPars_Ctrl_MS.in => FromPars.toNWK_Ctrl_MS,
FromPars_Ctrl Sl.out => FromPars.toNWK_Ctrl_Sl,

NWK_PANID_H.in => NWK_PANID_H,
NWK_PANID_L.in => NWK_PANID_L,
NWK_shortAddr_H.in => NWK_shortAddr_H,
NWK_shortAddr_L.in => NWK_shortAddr_L,
NWK_Parent_shortaddr_H.in => NWK_Parent_shortaddr_H,
NWK_Parent_shortaddr_L.in => NWK_Parent_shortaddr_L,

NWKDepth.in => NWKDepth,
MAC_state.in => MAC_state,
MS_Flag.in => MS_Flag,

LCD1.out => LCD1.out,
LCD2.out => LCD2.out,
```

--- test port ---

```vhdl
LED0_Search_Network => open,
LED1_Network_found => open,
LED2_Network_join => open,
LED3_Work.as.coordinator => open,
LED4_RoutingInfo.Sent => open,
LED5_NWK_data_receive => open,
LED6_NWK_data_sent => open,
LED7_TimePacket_sent => open
```

```vhdl
u6: ClockModule
  PORT MAP(
    CLKIN_IN => clock_48MHz,
    RST_IN => Reset,
    CLKDV_OUT => clock,
    CLkin_IBUFG_OUT => open,
    CLK0_OUT => open,
    LOCKEDOUT => open);
end structural;
```

### A.3 Radio Control Unit
A. VHDL SOURCE CODE

library ieee;
library ZigbeeLib;
use ieee.std.logic.1164.all;
use ieee.std.logic.arith.all;
use ZigbeeLib.zigbee.Constant.pkg.all;
use ZigbeeLib.zigbee.type.pkg.all;

entity TX_RX-Control.Unit is
port (  
clock : in std.logic;
resetn : in std.logic;

Data_to_parsing.out: out std.logic.vector (7 downto 0);
communication_control_to_parsing.out: out communication_control.Master.type;
communication_control_to_parsing.in: in communication_control.slave.type;

Data_from_Frame.in: in std.logic.vector (7 downto 0);
communication_control_from_Frame.in: in communication_control.master_type;
communication_control_from_Frame.out: out communication_control.slave.type;

Data_from_MAC.in: in std.logic.vector (7 downto 0);
Communication_control_from_MAC.in: in Communication_Control.master_type;
communication_control_from_MAC.out: out communication_control.slave.type;

Data_to_MAC.out: out std.logic.vector (7 downto 0);
Communication_Control_to_MAC.out: out Communication_Control.master_type;
Communication_Control_to_MAC.in: in communication_control.slave.type;

---** output ports to CC2420
SO: out std_logic;
SI: in std_logic;
SCLK: out std_logic;
CSn: out std_logic;

FIFO: in std_logic;
FIFOP: in std_logic;
CCA: in std_logic;

---
SFD: in std_logic;

--** test ports
  LED0.Receive.RXFIFO.Begin: out std_logic;
  LED1.Receive.RXFIFO.Complete: out std_logic;
  LED2.Access_TXFIFO: out std_logic;
  LED4.CCA.Clear: out std_logic;
  LED5.Done.Send.Packet: out std_logic;
  LED6.Send.Command.begin: out std_logic;
  LED7.Send.Command.Complete: out std_logic
--** end test port
end TX.RX.Control.Unit;

architecture New_behavior of TX.RX.Control_unit is
begin
  process (clock, resetn)
  --
  ***********************************************************************************************
  --** Define state machines' states. Each state machine is a enumeration
  **
  --** Variables are defined using the state machine type
  **
  --
  ******************************************************************************

  type SPIState.t is (SPI.IDLE, SPI.START, SEND.bit, RECEIVE.bit, SPI.STOP, Full.byte, BackToIdle);
  variable SPIState.v: SPIState.t;
  type Buff.state.t is (empty, with.data);
  variable Rx.buff.state.v, Tx.buff.state.v: buff.state.t;
  type SPI.User.t is (SPI.User.idle, RX.Control, TX.Control, Command.control);
  variable SPI.User.v: SPI.User.t;
type Receive_packet.State.t is (RX_idle,
    check_CC2420_state,
    RX_Req_SPI,
    Access_RX_FIFO,
    receive_CC2420_data,
    output_data,
    output_finish,
    receive.packet_complete,
    Clear_RX_FIFO,
    Clear_RX_FIFO.2,
    release_SPI_function,
    end.receive_packet);
variable Receive_packet.State.v: Receive_packet.State.t;

type send_packet.State.t is (TX_IDLE,
    TX_Req_SPI,
    Flush_Tx_FIFO.One,
    Flush_Tx_FIFO.two,
    Access_TX_FIFO,
    send.packet,
    finish_transfer,
    SendSTXONCCA,
    end.send.Packet,
    done.send.packet);
variable send_packet.State.v: send_packet.State.t;

type send.command.state.t is (Command_idle,
    Command_Req_SPI,
    start.send.command,
    send.command,
    LastByte.Feedback,
    finish.send.command);
variable send.command.state.v: send.command.state.t;

— SPI Data Registers
subtype char.t is std_logic_vector(7 downto 0);
variable Tx.v, Tx.Buff.v : char.t; — output buffer
variable Rx.v, Rx.Buff.v : char.t; — input shifter

— Counter Registers
subtype bit.count.t is natural range 0 to 7;
variable Bit.Count.v : bit.count.t;
variable byte.Count.v : natural range 0 to MaxFIFOSize.c;
variable byte.expect.v : natural range 0 to MaxFIFOSize.c;

— receive.mode.reg
variable MAC.to_TX_RX_interface.v : communication.Control_slave.type;
variable Frame.to.TX.RX.interface.v : communication.Control.slave.type;
—send_mode_reg
variable TX.RX.to.MAC_interface.v : communication.Control.master.type;
variable TX.RX.to.parsing_interface.v :
communication.Control.master.type;
variable TX.RX.to.parsing.Data.v : char.t;
variable TX.RX.to.MAC.Data.v: char.t;
variable SO.v: std.logic;
variable SCLK.v: std.logic;
variable CSn.v: std.logic;
— test variables
variable LED0.Receive.RXFIFO.Begin.v: std.logic;
variable LED1.Receive.RXFIFO.Complete.v: std.logic;
variable LED2.Access.TXFIFO.v: std.logic;
variable LED4.CCA.Clear.v: std.logic;
variable LED5.Done.Send.Packet.v: std.logic;
variable LED6.sent.Command.begin_v: std.logic;
variable LED7.Send.Command.Complete.v: std.logic;

procedure init.rcgs is
begin
SPIState.v:= SPI.IDLE;
SPI.User.v:=SPI.User_idle;
Send.packet.state.v:=TX.IDLE;
Receive.packet.state.v:=RX.Idle;
send.command.State.v:=Command.IDLE;
Rx.buff.State.v:=empty;
Tx.buff.state.v:=empty;
— MAC.to.TX.RX.interface.v := idle;
— Frame.to.TX.RX_interface.v := idle;
— TX.RX.to.MAC_interface.v := idle;
— TX.RX.to.parsing.interface.v := idle;
MAC.to.TX.RX.interface.v:=idle;
Frame.to.TX.RX.interface.v:=idle;
TX.RX.to.MAC.interface.v:=idle;
TX.RX.to.parsing.interface.v:=idle;
Bit.Count.v:=0; — receive bit
byte.Count.v:=0; — receive byte number
byte.expect.v:=0;
Rx.v:=(others => 'O');
Tx.v:=(others => 'O');
Rx.Buff.v:= (others => 'O');
Tx.Buff.v:=(others => 'O');
LED0.Receive.RXFIFO_Begin.v:= '0';
LED1.Receive.RXFIFO.Complete.v := '0';
LED2.Access.TXFIFO.v := '0';
LED3.Send_Command_Complete.v := '0';
LED4.CCA.Clear.v := '0';
LED5.Done.Send_Packet.v := '0';
LED6.sent.Command.begin.v := '0';
LED7.Send_Command_Complete.v := '0';

-- intend to be wire

end; -- procedure init.regs;

procedure update.ports is
begin -- purpose: synthesize a wire from the register to the port

Data.to.parsing.out <= TX.RX.to.parsing.Data.v;
communication.control.to.parsing.out <= TX.RX.to.parsing.interface.v;
communication.control.from.Frame.out <= Frame.to_TX.RX.interface.v;
communication.control.from.MAC.out <= MAC.to_TX.RX.interface.v;
Data.to.MAC.out <= TX.RX.to.MAC.Data.v;
Communication.Control.to.MAC.out <= TX.RX.to.MAC.interface.v;
SO<=SO.v;
SCLK<=SCLK.v;
CSn<=CSn.v;

LED0.Receive.RXFIFO.Begin<=LED0.Receive.RXFIFO_Begin.v;
LED1.Receive.RXFIFO.Complete<=LED1.Receive.RXFIFO.Complete.v;
LED4.CCA.Clear<=LED4.CCA.Clear.v;
LED5.Done.Send_Packet<=LED5.Done.Send_Packet.v;
LED7.Send_Command_Complete<=LED7.Send_Command_Complete.v;

end; -- procedure update.ports;

procedure SPI.state is
begin -- procedure SPI.state

case SPIState.v is

when SPLIDLE =>
SCLK.v:= '0';
CSn.v:= '1';
if Tx.buff.state.v = with.data then
SPIState.v := SPLSTART; -- kick off transmit
end if;

when SPLSTART =>
CSn.v := '0';
Bit.Count.v := 0;
byte.count.v := 0;

end case;

end SPI.state;
A. VHDL SOURCE CODE

```
Tx.v := TX.Buff.v;
Tx.Buff.state.v := empty;
SPIState.v := SEND.bit;

when SEND.bit =>
  SCLK.v := '0';
  SO.v := TX.v(Tx.v 'left);
  Tx.v := Tx.v(Tx.v 'left -1 downto 0) & '0';
  SPIState.v := receive.bit;

when receive.bit =>
  SCLK.v := '1';
  Rx.v := Rx.v(Rx.v 'left -1 downto 0) & SI;
  if bit.count.v = Rx.v 'left then
    bit.count.v := 0;
    byte.count.v := byte.count.v + 1;
    SPIState.v := full.byte;
  else
    bit.count.v := bit.count.v + 1;
    SPIState.v := send.bit;
  end if;

when full.byte =>
  SCLK.v := '0';
  if byte.count.v = byte.expect.v then
    SPIState.v := SPI.stop;
  elsif (Rx.Buff.state.v = empty) and (Tx.Buff.state.v = with.data) then
    Tx.v := TX.Buff.v;
    Tx.Buff.state.v := empty;
    Rx.Buff.v := Rx.v;
    Rx.Buff.state.v := with.data;
    SPIState.v := send.bit;
  end if;

when SPI.STOP =>
  CSn.v := '1';
  if Rx.Buff.state.v = empty then
    Rx.Buff.v := Rx.v;
    Rx.Buff.state.v := with.data;
    Tx.v := (others => '0');
    Rx.v := (others => '0');
    Tx.Buff.state.v := empty;
    SPIState.v := BackToIdle;
  end if;

when BackToIdle =>
  if Rx.Buff.state.v = empty then
    SPIState.v := SPI.idle;
  end if;

when others =>
  CSn.v := '1';
  SCLK.v := '0';
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
A. VHDL SOURCE CODE

```vhdl
procedure SPI.state is
begin
  case SPI_User_v is
    when SPI_User.idle =>
      if Receive_packet_State_v=RX.Rcq_SPI then
        SPI_User_v:=RX.Control;
      elsif send_command_state_v=Command.req_SPI then
        SPI_User_v:=Command_Control;
      elsif send.packet_State_v=TX.Rq.SPI then
        SPI_User_v:=Tx.Control;
      else
        SPI_User_v:=SPI_User_idle;
      end if;
    when RX.Control =>
      if Receive.packet_State_v=end.receive.packet then
        SPI_User_v:=SPI_User_idle;
      end if;
    when Command_control=>
      if send.command_state_v=finish_send.command then
        SPI_User_v:=SPI_User_idle;
      end if;
    when TX.control =>
      if send.packet.State_v=done.send.Packet then
        SPI_User_v:=SPI_User_idle;
      end if;
    when others=>
      SPI_User_v:=SPI_User_idle;
  end case;
end; -- procedure SPI.state;

procedure SPI.User-Control is
begin
  case SPI_User_v is
    when SPI_User_idle =>
      if Receive.packet.State_v=RX,Req_SPI then
        SPI_User_v:=RX.Control;
      elsif send.command_state_v=Command.req_SPI then
        SPI_User_v:=Command_Control;
      elsif send.packet.State_v=TX.Rq.SPI then
        SPI_User_v:=Tx.Control;
      else
        SPI_User_v:=SPI_User_idle;
      end if;
    when RX.Control =>
      if Receive.packet.State_v=end.receive.packet then
        SPI_User_v:=SPI_User_idle;
      end if;
    when Command_control=>
      if send.command_state_v=finish_send.command then
        SPI_User_v:=SPI_User_idle;
      end if;
    when TX.control =>
      if send.packet.State_v=done.send.Packet then
        SPI_User_v:=SPI_User_idle;
      end if;
    when others=>
      SPI_User_v:=SPI_User_idle;
  end case;
end; -- procedure SPI.User-Control;

procedure Receive.Packet is
begin
  case Receive.packet.State_v is
    when RX.idle =>
      if FIFO= '1' then
        Receive.packet_state.v := Check.CC2420.state;
      end if;
    when check.CC2420.state=>
      if SFD='0' and FIFO='1' then
        Receive_packet_state_v := RX.req.SPI;
      end if;
    when RX.Rq.SPI=>
      if SPI.User_v=RX.Control then
        Receive.Packet_state.v := Access.RX.FIFO;
```
A. VHDL SOURCE CODE

end if;
when Access.RX_FIFO=>
  if SPIState_v=SPI_idle then
    Tx.Buff.V:=RXFIFO_e; — constant
    TX.Buff.state_v:=with_data;
    byte.expect_v:=3;
    Receive_packet_State_v:=receive.cc2420.data;
    LED0.Receive.RXFIFO_Begin.v:= '1';
  end if;
when receive.cc2420.data=>
  if RX_buff.state_v=with.data and byte.count_v=1 then
    Rx缓冲状态_v:=empty; — ignore receive data;
  elsif RX_buff.state_v=full.byte and byte.count_v=2 then
    — update the byte expect to real length
    if Rx_buff_v/="00000000" then
      byte.expect_v:=Conv.integer(unsigned(Rx_buff.v))+2;
      Receive_packet_State_v:=output.data;
    else
      Receive_packet_State_v:=output.data;
    end if;
    end if;

  set x"00" in TX.buff, keep SPI have data send out
  if ((SPIState_v=send_bit) or (SPIState_v=receive_bit) or
      (SPIState_v=full.byte) and (Tx_buff.state_v=empty) then
    Tx_buff_v:="00000000";
    Tx.Buff.state_v:=with_data;
  end if;
when output.data=> — in this state, Rx.buff.State must be
  with.data

  check the change to send TX.Buff_v:=0;
  if RX_buff.state_v=with.data then
    TX_RX_to_parsing.Interface_v:=Rx.Buff.v;
    case communication.control_to_parsing.in is
      when idle=>
        TX_RX_to_parsing_interface.v:=send1;
      when receive1=>
        TX_RX_to_parsing_interface.v:=send2;
      when receive2=>
        TX_RX_to_parsing_interface.v:=send1;
      when others =>
        null;
    end case;
    Receive_packet_State_v:=output.finish;
  elsif RX_buff.state_v=empty and SPIState_v=SPI_idle then
    Receive_packet_State_v:=receive.packet_complete;
  end if;

  set x"00" in TX.buff, keep SPI have data send out
  if ((SPIState_v=send_bit) or (SPIState_v=receive_bit) or
      (SPIState_v=full.byte))
and (Tx_Buff.state_v=empty) then
    Tx.Buff.v:="00000000";
    Tx.Buff.state_v:=with.data;
  end if;
end if;

when output_finish =>
  if (TX.RX_to_parsing_interface.v=send1 and
      communication_control_to_parsing.in=receive1)
  or (TX.RX_to_parsing_interface.v=send2 and
      communication_control_to_parsing.in=receive2)
  then
    RX.Buff.State.v := empty;
    if SPIState.v=SPI_idle then
      Receive.packet.State.v := receive.packet.complete;
    else
      Receive.packet.State.v := output_data;
    end if;
  end if;
  
  set x"00" in TX.buff, keep SPI have data send out
  if ((SPIState.v=send_bit) or (SPIState.v=receive_bit) or
      (SPIState.v=full.byte)) and (Tx.Buff_state.v=empty)
  then
    Tx.Buff.v := "00000000";
    Tx.Buff.state.v := with.data;
  end if;

  when receive.packet.complete =>
    TX.RX_to_parsing_interface.v := idle;
    Receive.Packet.state.v := Clear.RX_FIFO;
    when Clear_RX_FIFO =>
      if SPIState.v=SPI_idle and Tx.Buff_state.v=empty
      and Rx_buff_state.v=empty then
        Tx.Buff.v := SFLUSHRX_c;  -- constant
        Tx.Buff.state.v := with.data;
        byte_expect.v := 1;
      end if;
      
      if Rx_buff_state.v=with.data then
        Rx_buff_state.v := empty;
      end if;
      
      if ((SPIState.v=send_bit) or (SPIState.v=receive_bit)
      or (SPIState.v=full.byte)) and (Tx.Buff_state.v=empty)
      then
        Tx.Buff.v := "00000000";
        Tx.Buff.state.v := with.data;
      end if;
      
      if (SPIState.v=SPI_Stop) then
        Receive.Packet.state.v := Clear_RX_FIFO_2;
      end if;
    end when Clear_RX_FIFO_2 =>
      if Rx_buff_state.v=with.data then
        Rx_buff_state.v := empty;
      end if;
      
      if SPIState.v=SPI_idle and Tx.Buff_state.v=empty
      and Rx_buff_state.v=empty then
A. VHDL SOURCE CODE

```vhdl
Tx.Buff.V := SFLUSHRX_c; — constant
TX.Buff.state.V := with.data;
byte.expect.V := 1;
end if;

if ((SPIState_V = send_bit) or (SPIState_V = receive_bit)
or (SPIState_V = full_byte)) and (Tx.Buff.state_V = empty) then
  Tx.Buff.V := "00000000";
  Tx.Buff.state.V := with.data;
end if;

if (SPIState_V = SPI_Stop) then
  Receive.Packet.state.V := release_SPI_function;
end if;
when release_SPI_function =>
  if Rx_buff_state_V = with_data then
    Rx_buff_state.V := empty;
  end if;
  if SPIState_V = SPI_idle and Tx.Buff.state_V = empty
  and Rx.Buff.state_V = empty then
    Receive.Packet.state.V := end_receive_packet;
  end if;
when end_receive_packet =>
  TX_RX_to_parsing_interface.V := idle;
  Receive.Packet.state.V := RX_idle;
  LED1.Receive.RXFIFO.Complete.V := '1';
when others =>
  Receive.Packet.state.V := RX_idle;
end case;
end;
```

```vhdl
procedure send.packet is
begin
  Case send_packet.State.V is
  when TX_IDLE =>
    if communication_control.from_Frame.in = send1 then
      send_packet.state.V := TX.req.SPI;
    end if;
  when TX_REQ_SPI =>
    if SPI.User.V = TX_Control then
      Send_Packet.state.V := Flush.TX_FIFO.One;
    end if;
  when Flush.TX_FIFO.One =>
    if SPIState_V = SPI_idle and Tx.Buff.state_V = empty
  and Rx.Buff.state_V = empty then
      Tx.Buff.V := SFLUSHTX_c; — constant
      TX.Buff.state.V := with.data;
      byte.expect.V := 1;
    end if;
  when Rx.Buff.state.V = with_data then
    Rx.Buff.state.V := empty;
end case;
end; — procedure receive_packet;

procedure send.packet is
```
end if;

if ((SPIState_v=send_bit) or (SPIState_v=receive_bit) or (SPIState_v=full_byte)) and (Tx.Buff_state_v=empty) then
  Tx.Buff_v="00000000"
  Tx.Buff_state_v:=with_data;
end if;

if (SPIState_v=SPI.Stop) then
  Send.Packet_state_v:=Flush_TX.FIFO.two;
end if;

When Flush.Tx.FIFO.two=>
if Rx.buff_state_v=with_data then
  Rx.buff_state_v:=empty;
end if;

if SPIState_v=SPI.idle and Tx.Buff_state_v=empty then
  Tx.Buff_v:=SFLUSHTX.c; -- constant
  TX.Buff_state_v:=with_data;
  byte.expect_v:=1;
end if;

if ((SPIState_v=send_bit) or (SPIState_v=receive_bit) or (SPIState_v=full_byte)) and (Tx.Buff_state_v=empty) then
  Tx.Buff_v="00000000"
  Tx.Buff_state_v:=with_data;
end if;

if (SPIState_v=SPI.Stop) then
  Send.Packet_state_v:=ACCESS.TX_FIFO;
end if;

when Access.TX_FIFO=>
if Rx.buff_state_v=with_data then
  Rx.buff_state_v:=empty;
end if;

if SPIState_v=SPI.idle and Tx.Buff_state_v=empty then
  Tx.Buff_v:=TXFIFO_c; -- constant
  TX.Buff_state_v:=with_data;
  byte.expect_v:=Conv_integer(unsigned(Data.from.Frame.in));
  send.Packet_state_v:=send_packet;
  LED2.Access.TXFIFO_v:=’1’;
end if;

when send.packet=>
if Rx.buff_state_v=with_data then --ignore receive package
A. VHDL SOURCE CODE

R.x_buff_state_v := empty;
end if;

if Tx.Buff_state_v=empty then
  case communication.control_from.Frame_in is
    when idle =>
      send.Packet.state_v := finish_transfer;
    when send1 =>
      if Frame_to_TX_RX_interface.v=idle or
         Frame_to_TX_RX_interface.v=receive2 then
        Tx_buff_v := Data_from_Frame_in;
        Tx.Buff.state_v := with_data;
        Frame_to_TX_RX_interface.v := receive1;
      end if;
    when send2 =>
      if Frame_to_TX_RX_interface.v=receive1 then
        Tx_buff_v := Data_from_Frame_in;
        Tx.Buff.state_v := with_data;
        Frame_to_TX_RX_interface.v := receive2;
      end if;
    when others =>
      null;
  end case;
end if;

when finish_transfer =>
  if ((SPIState_v=send_bit) or (SPIState_v=receive_bit)
    or (SPIState_v=full.byte)) and (Tx.Buff.state_v=empty) then
    Tx.Buff_v := '00000000';
    Tx.Buff.state_v := with_data;
  end if;
  if Rx_buff_state_v=with_data then
    Rx_buff_state_v := empty;
  end if;
  if (SPIState_v=SPI_Idle) then
    Frame_to_TX_RX_interface.v := idle;
    Send.Packet.state_v := send_STXONCCA;
    LED3.Send_Command_Complete.v := '1';
  end if;
when Send_STXONCCA =>
  if Rx_buff_state_v=with_data then
    Rx_buff_state_v := empty;
  end if;
  if (SPIState_v=SPI_idle) and (Tx.Buff_state_v=empty) and
    (Rx_buff_state_v=empty) then
    Tx.Buff_v := STXONCCA; -- constant
    TX.Buff_state_v := with_data;
    byte.expect_v := 1;
  end if;
if ((SPIState.v=send.bit) or (SPIState.v= receive.bit) or (SPIState.v=full.byte)) and (Tx.Buff.state.v=empty) then
Tx.Buff.v="00000000";
Tx.Buff.state.v=with.data;
end if;
if (SPIState.v=SPI_Stop) then
Send.Packet.state.v:=end.send.packet;
end if;
when end.send.packet=>
if Rx.buff.state.v=with.data then
Rx.buff.state.v=empty;
end if;
if SPIState.v=SPI_idle then
Send.Packet.state.v:=Done.send.packet;
Send.Packet.state.v:=TX_idle;
end if;
end case;
end; -- procedure send.packet;

procedure send.command is
begin
  case send.command.state.v is
    when Command_idle=>
      if Communication.control.from_MAC.in=send1 then
        send.command.state.v:=command_req_SPI;
      end if;
    when command_req_SPI=>
      if SPI>User.v=command.control then
        send.command.state.v:=start_send.command;
        byte.expect.v:= Conv.integer(unsigned(Data_from_MAC.in));
        MAC_to_TX_RX_interface.v:=receive1;
      end if;
    when start_send.command=>
      if (Communication.control.from_MAC.in=send2) and (MAC_to_TX_RX_interface.v=receive1) then
        Tx.Buff.V:=Data_from_MAC.in; -- constant
        Tx.Buff.state.v:=with.data;
        MAC_to_TX_RX_interface.v:=receive2;
      end if;
  end case;
end; -- procedure send.command;
A. VHDL SOURCE CODE

```vhdl
send_command_state.v:=send.command;
LED6.sent.Command.begin.v:= '1';
end if;

when send.command=>

if (((Communication.control_from.MAC.in=send1) and (MAC.to.TX_RX.interface.v=receive2))
or((Communication.control_from.MAC.in=send2) and (MAC.to.TX_RX.interface.v=receive1)))
then
  if(((SPIState.v=SEND_bit) or (SPIState.v=receive_bit) or (SPIState.v=full_byte))
  then
    Tx.Buff.V:=Data_from.MAC.in;  — constant
    TX.Buff.state.v:=with.data;
    case Communication.control_from.MAC.in is
      when send1=>
        MAC.to.TX.RX.interface.v:=receive1;
      when send2=>
        MAC.to.TX.RX.interface.v:=receive2;
      when others =>
        null;
      end case;
  end if;
  elseif (Communication.control_from.MAC.in=idle) and
  ((MAC.to.TX_RX.interface.v=receive1) or (MAC.to.TX_RX.interface.v=receive2))
  then
    MAC.to.TX.RX.interface.v:=idle;
  end if;

if (Rx.buff.state.v=with.data) and
  ((Communication.control_to.MAC.in=receive1 and TX_RX.to.MAC.interface.v=send1)
or((Communication.control_to.MAC.in=receive2 and TX_RX.to.MAC.interface.v=send2)
or((Communication.control_to.MAC.in=idle and TX_RX.to.MAC.interface.v=idle)) then
  TX_RX.to.MAC.Data.v:=Rx.buff.v;
  Rx.buff.state.v:=empty;
  case TX_RX.to.MAC.interface.v is
    when send1 =>
      TX_RX.to.MAC.interface.v:=send2;
    when send2=>
      TX_RX.to.MAC.interface.v:=send1;
    when idle=>
      TX_RX.to.MAC.interface.v:=send1;
    when others=>
      TX_RX.to.MAC.interface.v:=idle;
  end case;
end if;

if (SPIState.v=BackToIdle) then
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
A. VHDL SOURCE CODE

send.command.state.v := LastByte.Feedback;
end if;

when LastByte.Feedback =>

if (Rx.buff_state.v=with.data) AND
   ( (Communication.control.to.MAC.in=receive1 and
      TX.RX.to.MAC.interface.v=send1)
   or (Communication.control.to.MAC.in=receive2 and
      TX.RX.to.MAC.interface.v=send2)
   or (Communication.control.to.MAC.in=idle and
      TX.RX.to.MAC.interface.v=idle) ) then
   TX.RX.to.MAC.Data.v := Rx.buff.v;
   Rx.buff_state.v := empty;
   case TX.RX.to.MAC.interface.v is
      when send1 =>
         TX.RX.to.MAC.interface.v := send2;
      when send2 =>
         TX.RX.to.MAC.interface.v := send1;
      when idle =>
         TX.RX.to.MAC.interface.v := send1;
      when others =>
         TX.RX.to.MAC.interface.v := idle;
   end case;
elseif (Rx.buff.state.v=empty)
   and ((Communication.control.to.MAC.in=receive1 and
      TX.RX.to.MAC.interface.v=send1) or
   (Communication.control.to.MAC.in=receive2 and
      TX.RX.to.MAC.interface.v=send2))
then
   TX.RX.to.MAC.interface.v := idle;
end if;

if (Communication.control.to.MAC.in=idle and
    TX.RX.to.MAC.interface.v=idle) then
send.command.state.v := finish.send.command;
end if;
when finish.send.command =>
   Send.command.state.v := command.idle;
   LED7.Send.Command.Complete.v := '1';
when others =>
   Send.command.state.v := command.idle;
end case;
end; — procedure send.command;

procedure update_regs is
begin — purpose: call the procedures above in the desired order
   Receive.packet;
   send_packet;
   send.command;
   SPI_User.Control;
   SPI.state;
   Send.command.state.v := command.idle;
end case;
end; — procedure send.command;
--- Main part of program.
begin
  if resetn = '0' then
    initregs;
  elsif rising_edge(clock) then
    updatereg;
  end if;
  updateports;
end process; -- main;

end new_behavior;
Junsong Liao was born in Guanzhou, China in 1972. He received his B. Eng. degree in electrical engineering in 1993 from the South China University of Technology. After his graduation, he worked for China Mobile as a radio engineer for over ten years. He is currently a candidate in the electrical and computer engineering M.A.Sc. program at the University of Windsor and will graduate in Fall 2007.