A Unifying Interface Abstraction for Accelerated Computing in Sensor Nodes

Srikrishna Iyer

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

Master of Science
in
Computer Engineering

Patrick Schaumont, Chair
Yaling Yang
Sandeep Shukla

August 9, 2011
Blacksburg, Virginia

Keywords: Sensor networks, hardware-software co-design, hardware acceleration, fpga, microcontroller
Copyright 2011, Srikrishna Iyer

A Unifying Interface Abstraction for Accelerated Computing in Sensor Nodes

Srikrishna Iyer

(ABSTRACT)

Hardware-software co-design techniques are very suitable to develop the next generation of sensornet applications, which have high computational demands. By making use of a low-power FPGA, the peak computational performance of a sensor node can be improved without significant degradation of the standby power dissipation. In this contribution, we present a methodology and tool to enable hardware/software codesign for sensor node application development. We present the integration of nesC, a sensornet programming language, with GEZEL, an easy-to-use hardware description language. We describe the hardware/software interface at different levels of abstraction: at the level of the design language, at the level of the co-simulator, and in the hardware implementation. We use a layered, uniform approach that is particularly suited to deal with the heterogeneous interfaces typically found on small embedded processors. We illustrate the strengths of our approach by means of a prototype application: the integration of a hardware-accelerated crypto-application in a nesC application.

Grant Information

This thesis is supported by the National Science Foundation under Grant No. CCF-0916763. Any opinions, results and conclusions or recommendations expressed in this material and related work are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Related Articles . . . 3

1.3 Thesis Organisation . . . 4

2 Background **6**

2.1 Application Co-design . . . 6

2.1.1 NesC . . . 6

2.1.2 GEZEL . . . 7

2.2 SUNSHINE Simulator . . . 8

2.2.1 SUNSHINE System Components . . . 9

2.2.2 SUNSHINE Architecture . . . 11

2.3 Putting it All Together: The Complete Design Flow . . . 12

3 Communication Interface-Abstraction Architecture **14**

3.1 Motivation . . . 15

3.2 The Layered Channel Model . . . 16

3.2.1 Architecture Design Decisions . . . 16

3.2.2 Communication Interface-Independent Layer (CIL) . . . 18

3.2.3 Communication Adaptation Layer (CAL) . . . 18

3.2.4 Communication Presentation/Peripheral Layer (CPL) . . . 19

3.3 NesC Layers . . . 20

3.4 GEZEL Layers 22

4 Communication Library **24**

4.1 Software Implementation 25

4.1.1 SPI 25

4.1.2 UART 27

4.1.3 GPIO 30

4.2 Hardware Implementation 32

4.2.1 SPI 32

4.2.2 UART 33

5 VHDL Code Generation from GEZEL Ipblocks **35**

5.1 The fdlvhd tool 36

5.1.1 Step 1: Design the VHDL Code 36

5.1.2 Step 2: C++ Implementation 38

5.1.3 Step 3: Recompiling the Code Generator 39

6 Proof of Concept **41**

6.1 Application Design 41

6.2 Simulation Results 43

6.3 Prototype 44

7 Conclusion and Future Work **46**

A Communication Library Benchmarks **49**

A.1 Performance Evaluation 49

A.1.1 Benchmarking Platform 49

A.1.2 Throughput (cycles per byte) 50

A.1.3 CPU Load 51

SPI 51

UART 54

GPIO 56
A.2 Hardware Resource Utilization 58
A.2.1 SPI 58
A.2.2 UART 59

B VHDL Code Generation Source File for Block-RAM 60

C Catalog of Ipblocks with Code Generation Support 63

List of Figures

2.1 SUNSHINE architecture 10

2.2 Co-design environment with GEZEL and nesC code 13

3.1 The unified Channel abstraction and three layers of our communication interface architecture 17

3.2 Layers in nesC and their interfacing 21

3.3 Layers in hardware as GEZEL modules and their interfacing 22

3.4 FSM in CAL layer which performs synchronization 23

4.1 The HplAtm128SpiC component 28

4.2 The HplAtm128UartC component 30

4.3 The GPIO interface: MCU is the sender and ACU, the receiver 30

6.1 AES Application using SPI (schematic) 42

6.2 Sensor node prototype with AVR and FPGA boards 45

A.1 The platform used for generating benchmarks 50

A.2 Performance measure: polled mode 52

A.3 Performance measure: interrupt-driven mode 52

A.4 SPI: Graph showing absolute and effective cycles per byte 53

A.5 SPI: Graph depicting the CPU load (polled and interrupt-driven) 53

A.6 Performance measure: polled mode 54

A.7 Performance measure: interrupt-driven mode 54

A.8 UART: Graph showing absolute and effective cycles per byte 55

A.9 UART: Graph depicting the CPU load (polled and interrupt-driven) 55
A.10 Performance measure: polled mode 56
A.11 Performance measure: interrupt-driven mode 56
A.12 GPIO: Graph showing absolute and effective cycles per byte 57
A.13 GPIO: Graph depicting the CPU load (polled and interrupt-driven) 57

List of Tables

6.1 AES computational costs 44

Chapter 1

Introduction

1.1 Motivation

Sensor nodes, the building blocks of sensornets, consist of wireless embedded platforms fitted with sensors. The flexible part of a sensor node application is typically executed as software on a small embedded microcontroller unit (MCU). During execution, the sensor node software has to juggle a combination of sensor readings, data-packet transmissions, and other concurrency-intensive operations. To help the development of sensor node software, systematic support in the form of tiny, lightweight operating system was developed over a decade ago [1]. Such a resource control software remains very popular to this day to support concurrency and, more recently, post-deployment re-programming [2, 3].

Most of the time, of course, a sensor node does nothing and is in sleep mode to conserve battery energy. When it wakes up, it completes its job as quickly as possible, and goes back to sleep. The time required to complete a job is limited by the peak performance that can be delivered by the sensor node components. This applies to computational limits of the embedded processor, as well as to communication-bandwidth limits between peripheral components and the processor.

Our research is motivated by the observation that, under similar resource constraints, parallel implementations in hardware deliver better peak performance than sequential implementations on a small micro-processor [4]. Thus, by adding a Field Programmable Gate Array (FPGA) to the sensor node, and by re-mapping part of the sensor node application onto the FPGA, the sensor node application has a higher peak computational performance.

A good example of the increased computational needs in future sensornet applications may be found in healthcare [5]. Sensor nodes can continuously monitor vital metrics such as blood pressure level, glucose level, heart rate, and so on. This data can be logged or transmitted, but it obviously represents a privacy issue that may call for encryption. Furthermore, a healthcare sensor may need authentication to support the security policy of their environment. All of this results in strong cryptographic requirements, which in turn imply higher computational needs [6].

A possible downside of adding FPGAs to a sensor node is that the resulting node has more resources, all of which increase latent energy consumption. Thus, the sensor node may become less efficient at doing nothing. We believe that this issue is less critical then it appears to be. Recent generations of flash-based FPGA have dedicated support for ultra-low-power standby operation. For example, an Actel IGLOO AGL250 FPGA (250K system gates) has a standby current of 24 μA; a standard alkaline type AAA battery has sufficient energy to power the FPGA in standby for 6.7 years. Thus, although FPGAs may not be suitable for all types of sensornet applications, we believe they are an appropriate choice for those applications whose peak computational load is beyond the capabilities of current MCU-based sensor nodes.

Unfortunately, the current generation of sensor node development environments such as TinyOS [1] or Contiki [2] do not support multiple programmable components. These environments support sensor nodes that have a single, central MCU. The sensor application is expressed with a limited form of concurrency, and the hardware abstraction layer is ori-

ented towards fixed components with limited configuration capabilities, such as timers, A/D converters and UARTs.

To implement our vision of an FPGA-enabled sensor node, we need to expand the capabilities of current sensor node design environments. In this paper, we focus on the lack of design support for hardware/software interfaces. This is the first step towards an approach that integrates hardware/software co-design into the sensor node application design flow.

Our solution integrates nesC, a well-known programming language for sensor node application development [7], with GEZEL, a cycle-based hardware description language [8]. We demonstrate a layered interface between application software and application hardware. It captures the interaction between parallel entities in hardware and software using `send` and `receive` operations. Our communication protocol provides a systematic refinement of communications in MCU software or FPGA hardware, onto typical inter-chip communication interfaces such as SPI and UART. Hence, the sensor node developer can use convenient high-level semantics to express hardware/software communications.

We demonstrate the effect of migrating the compute-intensive encryption part of a sensor node application in nesC into a parallel hardware implementation in GEZEL. Furthermore, we also demonstrate a prototype implementation that combines a micro-controller board and an FPGA board.

1.2 Related Articles

Our work has been described in the following papers:

1. **S. Iyer** , J. Zhang, Y. Yang, and P. Schaumont, “A Unifying Interface Abstraction for Accelerated Computing in Sensor Nodes,” Electronic System Level Synthesis Conference (ESLsyn), 2011 , vol., no., pp.1-6, 5-6 June 2011 doi: 10.1109/ESLsyn.2011.5952296.

2. J. Zhang, Y. Tang, S. Hirve, **S. Iyer** , P. Schaumont, and Y. Yang, "A Software-Hardware Emulator for Sensor Networks," IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON), Salt Lake City, UT, USA, June 2011 (*won the best paper award*).

3. J. Zhang, Y. Tang, S. Hirve, **S. Iyer** , P. Schaumont, and Y. Yang, "SUNSHINE: A Multi-Domain Sensor Network Simulator," ACM SIGMOBILE Mobile Computing and Communications Review Volume 14, Issue 4, October 2010.

4. J. Zhang, Y. Tang, S. Hirve, **S. Iyer** , P. Schaumont, and Y. Yang, "SUNSHINE: A Platform-Agnostic Sensor Network Simulator," ACMMobicom, 2010 (Poster).

1.3 Thesis Organisation

The remainder of the thesis is organized as follows:

In Chapter 2, we discuss background material: a brief review of nesC and GEZEL, and a discussion of related work in the area of hardware abstraction for sensor node applications. We also describe our SUNSHINE tool, a three-tier simulator for sensor node applications.

Chapter 3 presents a layered interface model for the connection of sensor node applications in nesC to sensor node hardware modules in GEZEL, called the Communication Interface-Abstraction Architecture.

Chapter 4 provides a detailed description of a communication library implemented in nesC and GEZEL using the layered approach.

In Chapter 5 we talk about an automatic VHDL code generation mechanism for library modules in GEZEL.

Finally, in Chapter 6, we present our proof-of-concept implementation with a sample application, showing how all the ideas are put together to complete the design flow.

The last Chapter 7 concludes the paper.

Chapter 2

Background

In this chapter, we present a complete design flow for sensor nodes with three stages - application development, co-simulation and automatic code generation with prototyping. First, we provide a brief description of the language constructs used in the application development stage. Integration of the GEZEL kernel with an instruction-set simulator (ISS) forms the cycle-accurate co-simulation stage, which we discuss next. Finally, automatic code-generation completes the path to final implementation onto the sensor node hardware.

2.1 Application Co-design

We use GEZEL and nesC for describing hardware and developing software respectively. We briefly introduce these languages in this section.

2.1.1 NesC

NesC is a C-like programming language which reflects TinyOS's event-based concurrency model [7]. A nesC application is composed of `components` which are tied together using

bi-directional `interfaces`. The design of a component requires a specification, which enlists the interfaces it `provides` and `uses`, as well as an `implementation`, which incorporates the logic tied to those interfaces.

NesC distinguishes a `command` from an `event`. A `command` is a synchronous invocation (`call`) of one component's interface by another component. An `event` is an asynchronous call-back (`signal`) from one component to the other. The event interface is used to implement a split-phase mechanism when interacting with slow components. In a split-phase approach, commands are used to initiate an operation (e.g. send one byte through a UART), and events are used to mark their completion (e.g. the UART peripheral indicates completion).

2.1.2 GEZEL

GEZEL is a hardware description language that captures hardware behavior at the cycle-accurate level, using FSMD (Finite State Machine with Datapath) semantics [8]. GEZEL provides an extensible simulation kernel that simulates the descriptions, as well as a code generator to convert GEZEL descriptions into synthesizable VHDL.

The extensibility of GEZEL makes it particularly suited for our application. The GEZEL kernel has a mechanism to include foreign simulation engines, such as instruction-set simulators. At the language-level, a foreign simulator is captured using an `ipblock` construct, a black-box module with a pre-defined behavior. When simulating a black-box module, the GEZEL kernel invokes the foreign simulator. The co-simulation proceeds in lock-step and ensures that the hardware-software co-simulation remains causal.

For our application, we have integrated the SimulAVR simulator, an open-source, cycle-accurate simulator for the Atmel series of AVR processors. GEZEL instantiates an AVR core as an `ipblock`. The core module only specifies the type of MCU and the software application it executes. I/O interfaces, such as SPI, UART, and parallel ports, can be

```
ipblock avr1 {
  iptype "atm128core";
  ipparm "exec=spitest";      // AVR binary
}

ipblock avr_spi_port (
  out ss     : ns(1);         // SPI pins
  out sck    : ns(1);
  out mosi   : ns(1);
  in  miso    : ns(1)) {

  iptype "atm128port";        // component type
  ipparm "core=avr1";         // attached to
  ipparm "port=B";            // avr1, port B
  ipparm "pindir=xxxx0111";   // pin configuration
}
```

Listing 2.1: AVR with SPI interface modeled in GEZEL

included using additional `ipblock`. Listing 2.1 shows an example of an AVR core running a program `spitest`, together with an SPI interface. The `avr_spi_port` module shows the actual I/O pins available from the AVR MCU; GEZEL FSMD modules of the hardware can be connected to these pins to obtain an integrated hardware-software model.

2.2 SUNSHINE Simulator

In this section, we explain the integration of three simulators - hardware domain simulator GEZEL [8], software ISS SimulAVR [9], and the network domain simulator TOSSIM [10]. We call our integrated simulation environment the *SUNSHINE simulator* (Sensor Unified aNalyzer for Software and Hardware in Networked Environments) [11]. We first describe each component of the simulator and then, discuss the SUNSHINE architecture.

2.2.1 SUNSHINE System Components

1. *GEZEL*: In section 2.1.2, we explained GEZEL as a hardware description language. Here, we explain GEZEL as a cycle accurate simulator. In GEZEL, a platform is defined as the combination of a microprocessor connected to one or more dedicated hardware modules. GEZEL provides a hardware-software co-design environment that seamlessly integrates the hardware and software simulation domains at cycle-level. GEZEL has been used for hardware/software co-design of crypto-processors [12], cryptographic hashing modules [13], and formal verification of security properties of hardware modules [14]. GEZEL models can be automatically translated into a hardware implementation that enables a user to create his/her hardware, to determine the functional correctness of the custom hardware within actual system context and to monitor cycle-accurate performance metrics for the design.

   GEZEL is the key technology to enable a user to optimize the partition between hardware and software, and to optimize the sensor nodes architecture. With the support of GEZEL, the simulator can capture the software-hardware interactions and performance at cycle-level in a networked context.

2. *SimulAVR*: SimulAVR is an instruction-set simulator that supports software domain simulation for the AVR family of microcontrollers (MCUs) from Atmel, which are popular in sensor node platforms. SimulAVR provides accurate timing of software execution and can simulate multiple AVR MCUs simultaneously. We have integrated SimulAVR into the GEZEL hardware simulation engine in SUNSHINE, and through this integration, the detailed interactions between sensor hardware and software can be evaluated. Currently, SimulAVR does not support simulation of sleep mode or wakeup mode of sensor nodes. We have added sleep and wakeup schemes to provide simulation support for energy saving mode of sensor networks.

3. *TOSSIM*: TOSSIM is an event-based simulator for TinyOS-based wireless sensor net-

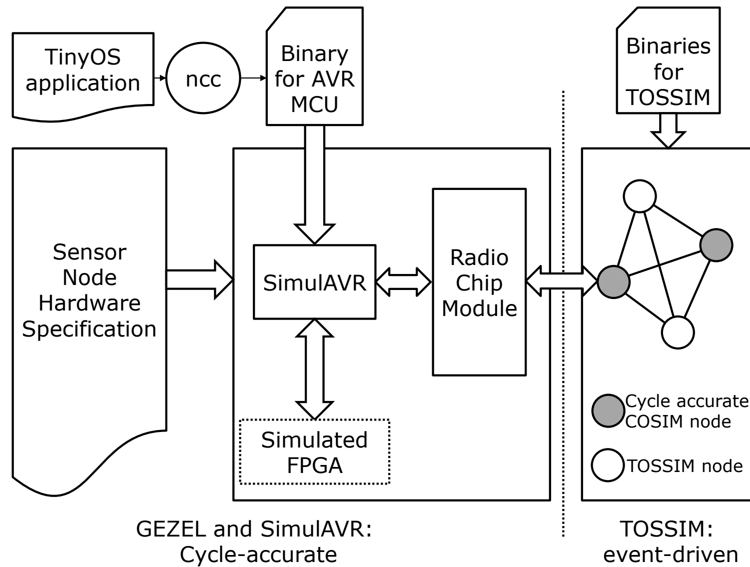

Figure 2.1: SUNSHINE architecture

works. TOSSIM essentially captures the network behaviors and interactions of a sensor node application. TOSSIM provides functional-level abstract implementations of both software and hardware modules for several existing sensor node architectures, such as the MICAz mote. In TOSSIM, an event-based network simulator, sensor node behaviors are regarded as functional-level events, which are kept in TOSSIMs event queue in sequence according to the event timestamps. These events are processed in ascending order of their timestamps. When the simulation time arrives at one events timestamp, that event is executed by the simulator. Even though TOSSIM captures the sensor motes behaviors and interactions, such as packet transmission, reception and packet losses at a high fidelity, it does not consider the execution time of the sensor node processor. Therefore, TOSSIM does not capture the fine-grained timing and interrupt properties of a software code.

2.2.2 SUNSHINE Architecture

SUNSHINE integrates TOSSIM, SimulAVR and GEZEL to simulate a sensornet in network, software, and hardware domains. A user of SUNSHINE can select a subset of sensor nodes to be emulated in hardware and software domains. These nodes are called cycle-level hardware-software co-simulated (co-sim) nodes and their cycle-level behaviors are accurately captured by SimulAVR and GEZEL. Other nodes are simulated in network domain by TOSSIM and only the high-level functional behaviors are captured. These nodes are called TOSSIM nodes. SUNSHINE can run multiple co-sim nodes with TOSSIM nodes in one simulation. The network topology in the right part of Figure 2.1 illustrates the basic idea of SUNSHINE. The white nodes are TOSSIM nodes, which are simulated in network domain, while the shaded nodes are co-sim nodes, which are emulated in software and hardware domains. When running simulation, these TOSSIM nodes and co-sim nodes interact with each other according to the network configuration and sensornet applications. Co-sim nodes simulate very slowly due the the high level of detail being captured. TOSSIM nodes simulate rapidly since only network events are captured. The mix of cycle-level simulation with event-based simulation ensures that SUNSHINE can leverage the fidelity of cycle-accurate simulation, while still benefiting from the scalability and speed of event-driven simulation.

The simulation process in SUNSHINE is illustrated by Figure 2.1. Firstly, for co-sim nodes that emulate real sensor motes, executable binaries are compiled from TinyOS applications using nesC compiler (ncc) and executed directly over these co-sim nodes. This is because co-sim nodes emulate hardware platform at cycle level. Therefore, TinyOS executable binaries can be interpreted by SimulAVR, the AVR simulation component of SUNSHINE, instruction-by-instruction. At the same time, GEZEL interprets the sensor nodes hardware architecture description, and simulates the AVR MCUs interactions with other hardware modules at every clock cycle. One of the hardware modules that GEZEL simulates is the radio chip module. This radio chip module provides an interface to TOSSIM, which models

the wireless communication channels. Through these wireless channels, co-sim nodes interact with other sensor nodes, which are simulated either as co-sim nodes by GEZEL and SimulAVR, or as functional-level nodes by TOSSIM. To maintain the correct causal relationship, the interactions between TOSSIM nodes and co-sim nodes are based on the timing synchronization and cross-domain data exchange techniques, which are explained in detail in [11].

For the remainder of this thesis, we will restrict our discussion to GEZEL-SimulAVR interactions and not consider TOSSIM further. For all intents and purposes, the SUNSHINE simulator will be used only as a hardware-software co-simulation tool in all further discussions, unless otherwise specified.

The SUNSHINE simulation environment along with the work presented in this paper is available for download at [15].

2.3 Putting it All Together: The Complete Design Flow

Figure 2.2 illustrates the complete design flow. The integration of the NesC language with GEZEL is the starting point of the design. Our target system architecture combines an FPGA and an AVR MCU. The FPGA-AVR interconnect is configurable and can use different interfaces of the AVR (SPI, UART, I^2C).

An application for this architecture is created as a combination of a software program in nesC with a platform description in GEZEL. The nesC program is compiled for AVR using the `ncc` compiler.

The SUNSHINE simulator parses the GEZEL description, instantiates an AVR instruction-

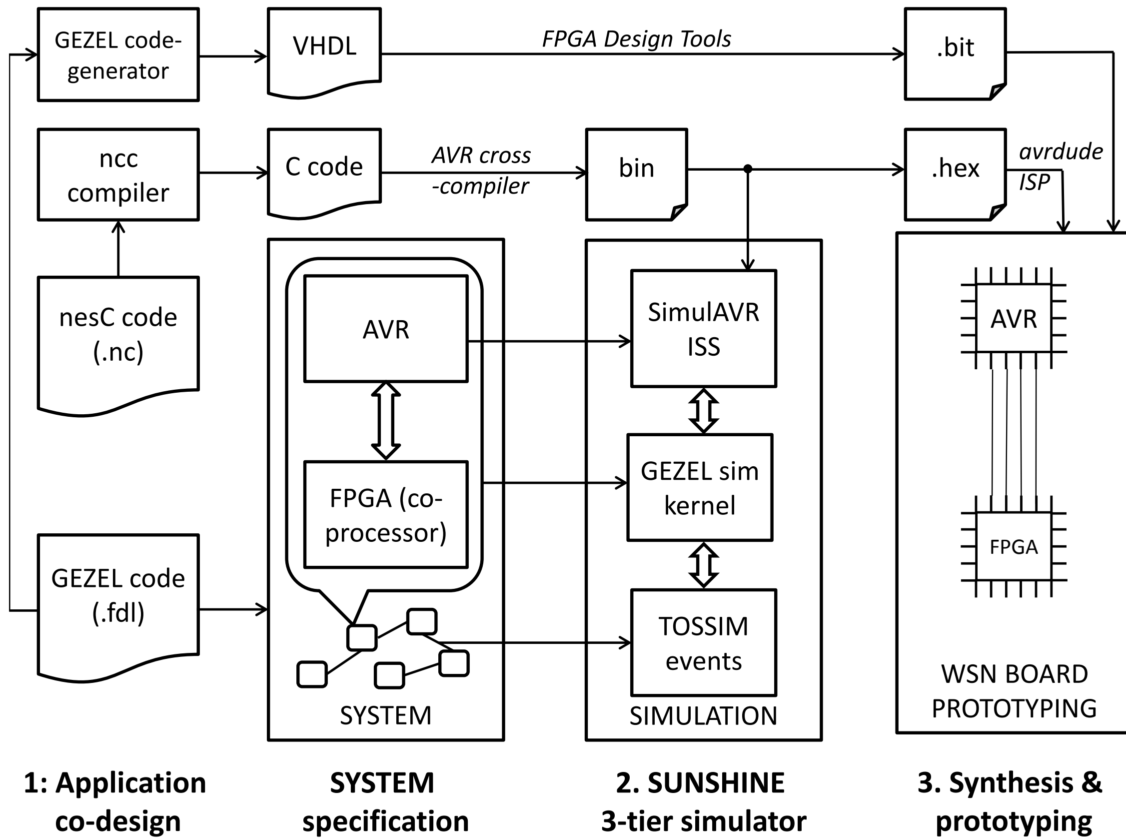

Figure 2.2: Co-design environment with GEZEL and nesC code

set simulator, and loads the NesC binary. A cycle-accurate simulation then enables the designer to predict the performance of the resulting system.

Automatic code generation follows the simulation stage. Here the binary which is to be run on the MCU and the VHDL code for the hardware design to be synthesized on the FPGA are generated. The resulting bitstream, as well as the nesC binary, can then be downloaded onto an actual sensor node prototype.

In the next chapter, we focus on the design and implementation of a structured communication channel between the MCU software and the FPGA hardware.

Chapter 3

Communication Interface-Abstraction Architecture

In the last chapter, we proposed a design flow for the development, simulation and the deployment of applications partitioned in hardware and software. We discussed an environment which allows co-design by integrating the GEZEL language and the nesC language. It also provides a multi-domain simulation framework and path to final implementation.

In this chapter, we will discuss the Communication Interface-Abstraction Architecture, which is the main focus of this thesis. It is an attempt at building an efficient hardware-software interface for a constrained, MCU-oriented environment. The main goal of designing this architecture is to provide easy connectivity between hardware and software parts of the application at a high level of abstraction, without having to deal with low level interactions. The software and hardware parts send and receive messages via the Communication Abstraction Architecture. It decomposes the messages into three layers of abstraction.

We will first discuss the need building such an architecture and the benefits it will provide while integrating hardware and software. Before describing the functions of the three layers

and their implementation in software and hardware in detail, we will discuss some design choices we made while designing the architecture.

3.1 Motivation

Our objective is to use an FPGA to accelerate compute-intensive tasks. Such a task, mapped in hardware, will be called an Accelerating Computation Unit (ACU). The ACU assists the MCU at completing the overall system activity in a shorter time.

MCUs typically offer many different interfaces which have a wide range of capabilities. Besides general purpose input/output pins (GPIO), this includes for example, UART (up to 250 Kbps), I^2C (400 Kbps), SPI (1 Mbps), as well as more advanced interfaces including CAN (250 Kbps), Ethernet (10 Mbps), and/or USB (1.5 Mbps). Serial interfaces are most common because they reduce the pin-count of the MCU package. A complete sensor node architecture can be integrated using these interfaces: all chips of a sensor node board (RAMs, ROMs, radio, sensors) are interconnected through a heterogeneous collection of MCU interfaces.

Within this environment, we need to integrate an ACU in such a way as to provide the most flexible platform for the application developer. When integrating hardware and software with a chosen interface, the ACU needs to implement a *driver* layer specific to that interface. Due to the lack of a standard way of combining the core ACU application with such a driving module, the ACU design remains inflexible. The need to develop a low level driver module for each core ACU application is tedious and decreases productivity when integrating hardware and software. Another challenge is the heterogeneity of interfaces, which may hamper the portability of applications written for the ACU. For the same ACU core, it may be possible that a hardware interface selected in one application may not be available in the other. Indeed, each time a different hardware interface is selected, the ACU application must be potentially rewritten.

These challenges led us to design a communication architecture which is described next.

3.2 The Layered Channel Model

The objective of our layered communication interface is to simplify the development of applications, for the hardware as well as for the software. Ideally, we aim to unify all possible communication schemes between an MCU and an ACU into a single higher level abstraction, called a `Channel`. The `Channel` provides a standard message level interface for attaching the ACU with the MCU. By separating the interfacing from the core application, co-design is a lot more easier and streamlined. Software, and the ACU need to deal only with message level interfaces. Also, due to modular approach, changing ACU-MCU interface translates to a simple replacement of interface specific layers in the communication architecture without changing the ACU or the MCU application itself. Since the ACU now only has to deal with high level message interface provided by the communication architecture, it has to be designed only once, and it will work for any interface; and likewise in the software side.

3.2.1 Architecture Design Decisions

We will first explain two interface design decisions before introducing the `Channel` abstraction.

The first design decision is to configure ACU and MCU as Slave and Master devices, respectively across the `Channel`. This implies that communication primitives, regardless of their direction (send/receive), are always initiated by the MCU. When the ACU needs to return a result to the MCU, it will remain idle until instructed by the MCU to return the result.

The second design decision is to use a synchronous model of communication for the `Channel`. This means that communication activities are executed in-line with other computational

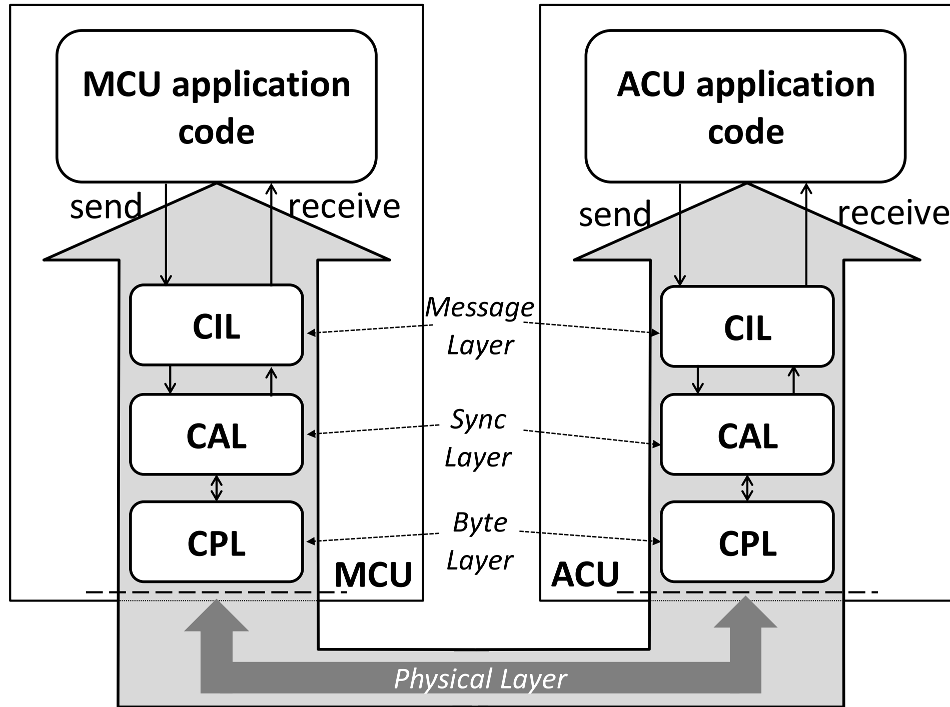

Figure 3.1: The unified Channel abstraction and three layers of our communication interface architecture

tasks. Considering the accelerated-computing model, hardware-software communication in polled mode makes more sense than an asynchronous interrupt-driven mode. First, the interrupt-driven model suffers from the overhead of frequent context switching. Next, hardware interrupts are a scarce resource on an MCU, and we choose to reserve them for truly asynchronous events such as messages received from the radio and sensor readings. Finally, for many applications, the computation time of the ACU is known, or it can be accurately measured using our co-simulator. Therefore, the MCU can select an optimal polling rate, and at the same time still benefit from sleep mode while the ACU is performing long computations.

Figure 3.1 illustrates the `Channel` abstraction between MCU and ACU. The layered abstraction ensures that the messages are correctly delivered at the other side for use, irrespective of the interface being used. Each abstraction layer of the `Channel` is implemented inside the

MCU as well as in the ACU. The topmost layer is independent of the communication interface used to connect MCU and ACU. The top layer offers simple send and receive primitives that support transmission of variable-length messages.

As we move towards the bottom of the layered interface, the intricacies of the underlying communication peripheral become increasingly visible. The lowest layer of the `Channel` corresponds to the physical wiring between the MCU and ACU.

In this perspective, our communication stack is not limited to the interconnection of hardware and software; the ACU may as well be another MCU connected as a slave device. In the next subsections, we briefly summarize the objective of each layer. After that, we describe their implementation in software (nesC) as well as in hardware (GEZEL).

3.2.2 Communication Interface-Independent Layer (CIL)

The **CIL** provides interface-independent functionality to `send` and `receive` messages to the main application code; hence it is also known as the *message layer*. The application code on the MCU and ACU will remain the same regardless of the underlying implementation of the `Channel`. This greatly simplifies application design.

The main function of this layer is to perform *message segmentation* in the `send` direction and *message aggregation* in the `receive` direction. Messages sent by the application code are split into equal-length packets, before being forwarded to the next lower layer. Similarly, the packets received from the middle layer are aggregated into complete meaningful messages which can be used by the application code above.

3.2.3 Communication Adaptation Layer (CAL)

CAL is also called *packet-level synchronization and multiplexing layer*. It has two functions.

The first function is to provide logical synchronization. The MCU and ACU operate in parallel and are asynchronous with respect to each other. Hence, this layer synchronizes the MCU and ACU at the packet level, so that correctness of the flow of data is maintained.

At runtime, the MCU does not know if the ACU can accept a message of arbitrary length. Hence, transferring an entire message involves the risk of dropped message bytes leading to a potential deadlock. Hence, we designed CAL to handle messages chopped into fixed length packets along with a handshake mechanism. Before any packet transfer, the MCU sends a request byte to the ACU. The request indicates the intended direction of communication. The ACU sends back a response byte indicating whether it can be completed. After the handshake, the entire packet transfer is done in burst. This in-band approach to handshaking comes at the cost of two extra byte transfers per packet. The alternative is to perform out-of-band handshaking using dedicated GPIO pins on the MCU. This would lessen the delay overhead, but it increases the signaling complexity and consumes a precious MCU resource. Our solution finds motivation from simplicity of implementation and minimal GPIO resource usage.

The second function is to distinguish between upstream and downstream data. Some, though not all, serial interfaces for MCUs can only operate in a bi-directional mode: for each bit sent, another bit is received. Since the upstream and downstream communications can be logically independent, additional de-multiplexing must be provided.

3.2.4 Communication Presentation/Peripheral Layer (CPL)

CPL forms the lowest *byte layer* in our abstraction, positioned right above the communication hardware peripheral. This abstraction 'presents' the hardware and provides easy to use functionality to access the hardware via a byte interface. The communication peripheral does the task of converting data stream from the physical layer to byte-stream onto the byte

```
1  interface MessageStream {
2    command result_t send(uint8_t *msg, uint16_t length);
3    event void sendDone(uint8_t *msg, uint16_t length, error_t error);
4    command result_t receive(uint8_t *msg, uint16_t length);
5    event void receiveDone(uint8_t *msg, uint16_t length, error_t error);
6  }
7
8  configuration ChannelC {
9    provides interface MessageStream;
10 }
11 implementation {
12   components CilUartC as HWCHannel;
13   MessageStream = HWCHannel;
14 }
```

Listing 3.1: CIL interfaces to the main application

interface and vice versa. In case this interface would be absent, then the CPL layer offers the possibility to emulate this functionality.

3.3 NesC Layers

NesC implements the hardware abstraction model of TinyOS, after which our layering is closely modeled [16]. NesC distinguishes three layers of hardware abstraction: a hardware-independent layer (HIL), a hardware-abstraction layer (HAL), and a hardware-presentation layer (HPL). The main goal of these layers is to expose the functionality of a fixed hardware which is independent of the sensor node platform. Our implementation, however, is aimed at providing a method to communicate with an external, re-configurable ACU and an interface-independent way of accessing it.

It is straightforward to express the layers of our communication abstraction for each interface in NesC. However. we found that the highest layer, HIL, still exposes information about the kind of communication peripheral used. In order to turn HIL interfaces into a completely generic CIL, we wrap the HIL implementation of the communication peripherals with a new component called `ChannelC`.

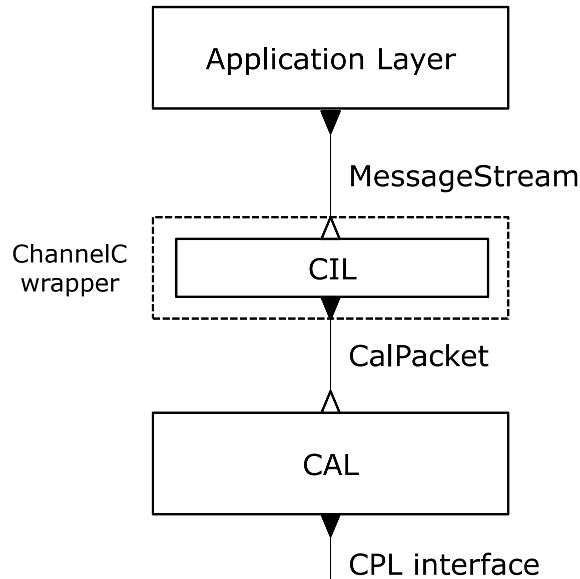

Figure 3.2: Layers in nesC and their interfacing

The `ChannelC` component provides an interface to the main application called `MessageStream` as shown in Listing 3.1. The main application uses this interface for message transfers. `ChannelC` component simply passes messages between the internal HIL and the `MessageStream`. This is illustrated in Figure 3.2.

The only purpose of `ChannelC` is to isolate the main application from the communication device-oriented HIL semantics. Applications thus developed will not only be platform-independent, but also, the actual hardware peripheral used for communication will be transparent leading to a truly generic message-passing scheme. In Listing 3.1, line 12 indicates that the ChannelC wraps around the CIL layer of UART called `CilUartC`.

The lower layer, CAL provides an interface `CalPacket`, which handles transmission of messages chopped down to a fixed packet size. The `PACKET_SIZE` is an application-dependent setting used by the CIL and CAL. All communication methods integrated into the architecture use `CalPacket` as the CAL interface to CIL.

The bottom layer, CPL is communication device-specific. We explain the CPL implementations of two such specific communication peripherals - SPI and UART in Chapter 4.

3.4 GEZEL Layers

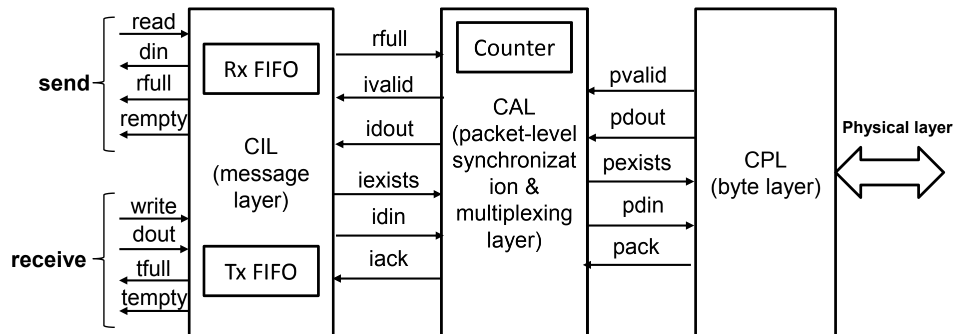

Figure 3.3: Layers in hardware as GEZEL modules and their interfacing

The application part mapped onto the ACU has a similar layered model. Figure 3.3 represents the interfaces provided to the GEZEL application.

The uppermost **CIL** layer is the only layer that stores data. It uses a FIFO in each of the transmit and receive directions to hold one packet. The application hardware in the ACU will access these FIFO's through a streaming interface, shown on the left side of Figure 3.3. The FIFO's need to be able to hold at least one packet, but they may be over-dimensioned depending on the application.

The **CAL** layer in hardware inherits the same streaming interface as the CIL layer. CAL maintains synchronization with the help of a finite state machine (FSM) and a byte counter. The FSM intercepts the request byte from the MCU. In response, it sends back a response byte by checking the FIFO signals as shown (symbolically) in Figure 3.4. Based on the request, the upstream, downstream or both are enabled. The byte counter tracks the correct number of bytes per packet. As soon as the set packet size is reached, the CAL resets to the idle state.

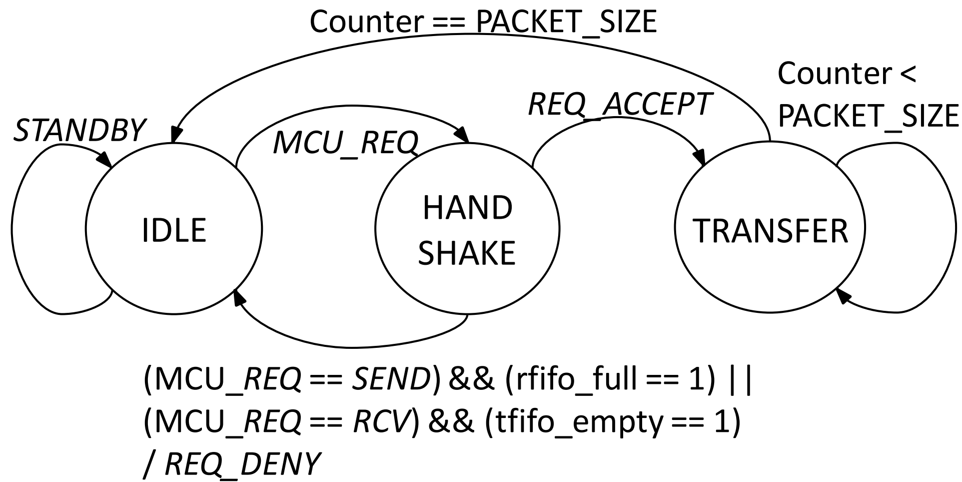

Figure 3.4: FSM in CAL layer which performs synchronization

The reconfigurable nature of FPGA obviates the need for simple dedicated hardware such as communication interfaces. Hence, in hardware, we implement an RTL description of the communication peripheral itself in the CPL layer. It converts the raw data stream on the physical layer into a byte stream for the CAL layer and vice versa. A standard set of streaming interfaces connects CPL with CAL as shown in Figure 3.3.

All abstraction layers in hardware (FPGA) are instantiated in the GEZEL application using the `ipblock` mechanism. This allows a user to quickly assemble the required communication stack in between a user-defined hardware application and a microprocessor interface (SPI, UART, I^2C).

Chapter 4

Communication Library

In this chapter, we will discuss the development of a communication library based on the Communication Abstraction Architecture described in the previous chapter. A typical MCU provides a bunch of interfaces to which an ACU can be attached. We have integrated three of such interfaces - SPI, UART and general purpose I/O (GPIO) pins within our architecture. Selection of an interface for attaching an ACU really depends on the application. One application may allow only a serial interface if there is a constraint on the number of port pins used to connect the ACU. On the other hand, if there is a high bandwidth requirement, a parallel interface is more suitable. Hence, apart from their implementation in software as well as in hardware, we will compare their relative performance based on the measured throughput. Also, we will compare their footprint in software in terms of code size and in hardware in terms of gate count. All these performance evaluation results can be found in Appendix A.

4.1 Software Implementation

We have integrated three communication interfaces into our architecture - SPI, UART and GPIO. Our implementation is specific to ATMega128 from Atmel, an MCU commonly used in wireless sensor network platforms. The reason for selecting these interfaces is to compare their relative performances in terms of throughput and the resource usage in terms of number of MCU port pins required for connectivity. The first two are industry-standard serial interfaces, SPI and UART. They are quite popular and are used to connect peripherals such as flash based RAM, radio chips and other devices. Due to their serial nature, they offer low resource usage at the cost of limited bandwidth. The third interface, GPIO is a custom built interface and has a variable point-to-point parallel connectivity. It has a higher resource usage compared to the serial interfaces. However, it provides an increased bandwidth due to parallel transfer.

We already discussed the top two layers of the Communication Abstraction Architecture in the previous chapter. In this chapter, we will discuss the communication peripheral-specific implementation which is in the CPL.

4.1.1 SPI

The SPI peripheral allows high speed synchronous data transfer between the MCU and external peripherals. It consists of four pins - SS (slave select), SCK (serial clock), MOSI (master out, slave in) and MISO (master in, slave out). The MCU can operate in either Master mode or Slave mode. When operating as Master, it pulls the SS of the desired slave low to activate it. Bit-wise data transfer on the MISO and MOSI lines occur simultaneously on the SCK clock provided by the SPI Master.

TinyOS 2.1.1 already provides the Presentation Layer component in nesC called HplAtm128-SpiC. It provides the interface `Atm128Spi`. The `Atm128Spi` interface provides functionality

to completely configure and run the SPI peripheral within the MCU. Some of its main commands are described in the Listing 4.1.

```
interface Atm128Spi {
  async command void initMaster();
  async command void initSlave();
  async command void enableInterrupt(bool enabled);
  async command void enableSpi(bool busOn);
  async command void sleep();
  async command void wakeup();
  async command uint8_t read();
  async command void write(uint8_t data);
  async event   void dataReady(uint8_t data);
  async command void  setClock(uint8_t speed);
  async command void  setMasterDoubleSpeed(bool on);
  // ...
}
```

Listing 4.1: Atm128Spi interface

`void initMaster()` & `void initSlave()`: The directions of the port pins associated with the SPI peripheral are initialized, depending on whether SPI is being configured as Master or slave.

`enableInterrupt(bool enabled)`: This command allows SPI to run in interrupt mode. It will cause the SPI peripheral to signal an interrupt at the end of a byte transfer.

`void enableSpi(bool busOn)`: By calling this command, the I/O port pins associated with SPI will be be reserved exclusively for SPI operations. The SPI peripheral will not be operational unless this command is called.

`void sleep()`: This command (available only when configured as Master) pulls the SS pin high causing the slave peripheral to be disabled.

`void wakeup()`: This command (available only when configured as Master) pulls the SS pin low causing the slave peripheral to be activated.

`void write(uint8_t data)`: The data to be sent out on the SPI bus is passed as an argument to this command.

`uint8_t read()`: At the end of SPI transfer, the data received from the external peripheral is returned.

`void dataReady(uint8_t data)`: When SPI is run in interrupt-driven mode, this function is evoked when interrupt is generated. This event is to be implemented by the *user* of the Atm128Spi interface.

`void setClock(uint8_t speed)` & `async command void setMasterDoubleSpeed(bool on)`: The two functions together decide at what frequency the SPI transfers occur. It can be anything from F_CPU/2 to F_CPU/128, where F_CPU is the MCU oscillator frequency.

The CAL component uses this interface to perform packet level transactions. The Figure 4.1 shows the CPL Layer and the associated interfaces wired to other components.

Also, applications using the SPI interface to connect an external ACU require two SPI-based settings:

1. `extern const uint8_t SPI_MODE`: SPI can be run in master or slave modes. This is set to 1 for master mode and 0 for slave mode.

2. `extern const uint8_t SPI_FREQUENCY`: This is the frequency at which SPI is to be run. Valid settings are FoscBy2, FoscBy4, FoscBy8, FoscBy16, FoscBy32, FoscBy64 and FoscBy128. The CPU frequency is prescaled by the indicated values and supplied as the SPI frequency.

4.1.2 UART

UART is a widely used serial communication device. It uses only two wires - TXD and RXD for transmitting and receiving data. The two peripherals communicating through UART need to have the same baud rate setting for the data to be transmitted correctly. The two UART peripherals in ATMega128 MCU are highly customizable. It supports variable data

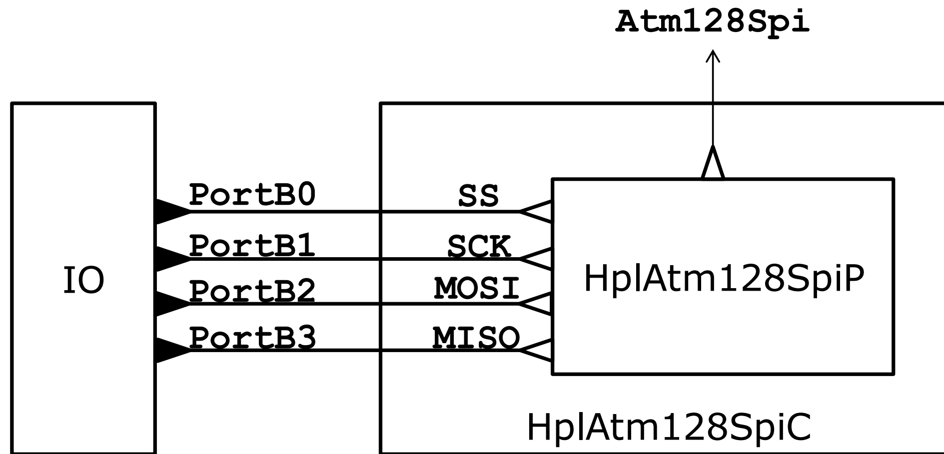

Figure 4.1: The HplAtm128SpiC component

length, odd and even parity, synchronous mode (an extra pin is required), error detection and double speed mode.

TinyOS 2.1.1 also includes an implementation of the UART Presentation Layer, which we reuse while integrating into our architecture. The HplAtm128UartC component provides the interface HplAtm128Uart which is used to access the UART peripheral. The HplAtm128Uart interface is shown in Listing 4.2.

```
1  interface HplAtm128Uart {
2    async command error_t enableTxIntr();
3    async command error_t disableTxIntr();
4    async command error_t enableRxIntr();
5    async command error_t disableRxIntr();
6    async command bool isTxEmpty();
7    async command bool isRxEmpty();
8    async command void tx( uint8_t data );
9    async event void txDone();
10   async command uint8_t rx();
11   async event void rxDone( uint8_t data );
12 }
```

Listing 4.2: HplAtm128Uart interface

`error_t enableTxIntr()` & `error_t disableTxIntr()`: These commands are respectively, used to enable and disable the generation of an interrupt when a byte is sent out.

`error_t enableRxIntr()` & `error_t disableRxIntr()`: Similarly, these commands enable/disable interrupt on reception.

`bool isTxEmpty()`: This commands checks whether an ongiong transmission is complete. If there is a pending transmission, this function will return false.

`bool isRxEmpty()`: This command checks whether a byte is currently being received or has been received. If there is no pending reception or if the UART receive register has been read out, this command will return false.

`void tx( uint8_t data )`: A byte to be transmitted is passed as an argument to this command. When the transmission is complete, an interrupt will be signaled (if enabled), and isTxEmpty() will return true. To ensure there is no write collision, always call isTxEmpty() before calling this command.

`event void txDone()`: This event is executed by the interrupt service routine associated with UART transmission. It has to be implemented by the *user* of this interface.

`uint8_t rx()`: This command will return the byte received.

`void rxDone( uint8_t data )`: This event is executed by the ISR associated with UART reception. It also has to be implemented by the *user* of this interface.

The Figure 4.2 shows the CPL of the UART peripheral. Initialization of the UART peripheral is done with the help of the `UartInit` interface. Is is tied to the `SoftwareInit` interface of the `MainC` component, which is the entry point of the entire application. This ensures that the UART initialized before any component in the nesC application uses it. In addition, the `UartTxControl` and `UartRxControl` are interfaces provided to the CAL to start and stop the UART transmission and reception respectively.

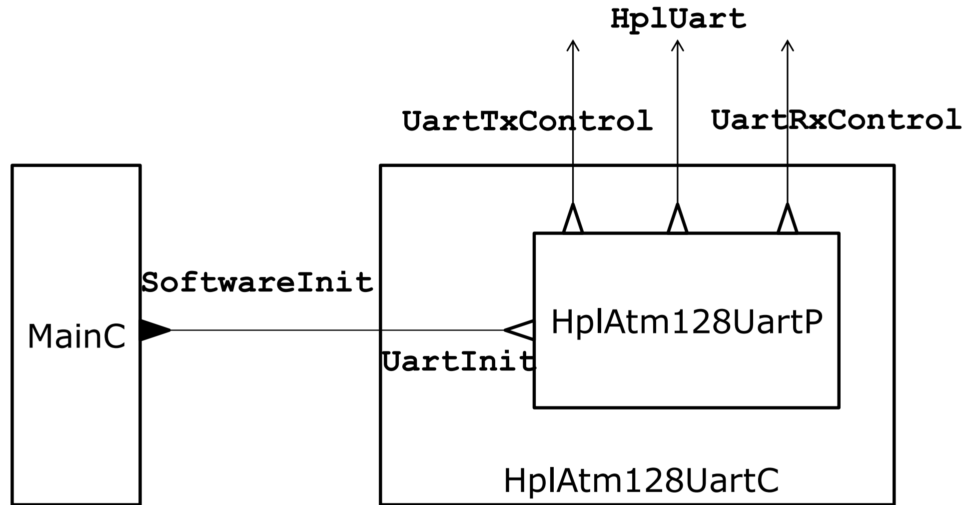

Figure 4.2: The HplAtm128UartC component

4.1.3 GPIO

The GPIO interface is a custom built interface in which the width of the data bus is variable. Currently, the implementation is simplex. Hence, data can be transmitted only in one direction. To reverse the direction, the I/O pin directions have to be changed. We chose to avoid duplex mode of communication purely because of the high pin resource usage. Also, the GPIO interface is currently limited to a C implementation, although, a nesC version is being planned in the future work. Apart from the number of variable data pins, two additional

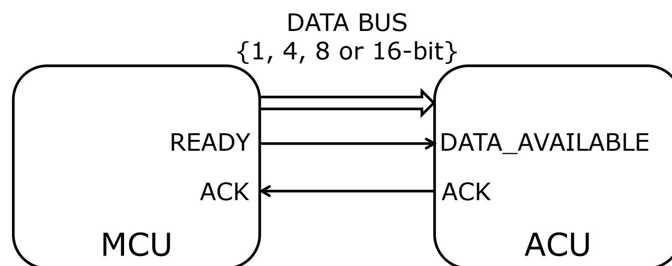

Figure 4.3: The GPIO interface: MCU is the sender and ACU, the receiver

handshake pins are used as shown in Figure 4.3. When transmitting data, they are called READY (output) and ACK (input). When receiving, they are called DATA_AVAILABLE

(input) and ACK (output). The same port pins are used for handshake during transmission and reception. Only their purpose is different. In interrupt driven mode, the handshake input in either case (ACK while sending and DATA_AVAILABLE while receiving) is connected to an external hardware interrupt pin.

The sender puts the data to be transmitted on the data bus and asserts the READY signal. The receiver of the GPIO interface detects the READY signal, accepts the data and asserts the ACK signal. In response, the sender de-asserts the READY signal and then, the receiver lowers the ACK signal. The receiver, on the other hand, waits for the DATA_AVAILABLE signal to be raised high. When that happens, it accepts the data on the data bus and asserts the ACK signal. The ACK is lowered when the DATA_AVAILABLE signal is de-asserted.

Currently, the implementation of GPIO is limited to a C implementation. The following is a list of associated functions:

`void portioInit(uint8_t direction, width_t width, uint8_t mode)`: This function initializes the GPIO transfer operation. The first argument is the direction: 1 for sending and 0 for receiving. The second arg is the width of the data being transferred at a time. The width argument supports the following sizes: BIT (single bit), NIBBLE (4 bits), BYTE (8 bits) and WORD (16 bits). As many I/O pins are initialized as the width, apart from the two handshake pins. The third argument decides whether to use interrupts (1 - interrupt-driven, 0 - polled). Note that this function needs to be called when the data transfer direction is to be changed.

`void portioSend( uint8_t *data )`: This function is used to transmit data. A single call to this function will transmit at least one byte of data at the location pointed by *(data) if the data width selected is less than or equal to a BYTE. If the data width selected is WORD, then a single call to this function will furnish two bytes from the locations *(data) and *(data + 1) and they will be transmitted in parallel. This function should hence be carefully used with 16-bit parallel transfer.

`void portioReceive( uint8_t *data )`: This function is used to receive data on the GPIO interface. This function operates exactly like the send function described above, only with the direction reversed. Received data is written to the location pointed by *(data).

4.2 Hardware Implementation

In hardware, we have implemented only the serial interfaces SPI and UART. As discussed in the last chapter, the CIL for all interfaces in implemented as a datapath. Its implementation is universal. Integration of an interface then requires the implementation of CAL and CPL. For both the interfaces, we have implemented the CAL and CPL as ipblocks. Unlike datapaths, ipblocks offer flexibility through the use of *ipparm* statements. Also, the GEZEL code size reduces to black-box instantiations.

4.2.1 SPI

The following listings are actual codes for CPL and CAL of the SPI interface. The CPL

```
1  ipblock spi_pl (
2    // spi ports
3    out miso   : ns(1);
4    in mosi    : ns(1);
5    in sck     : ns(1);
6    in ss      : ns(1);
7    // cal interface ports
8    out valid  : ns(1);
9    out dout   : ns(8);
10   in exists  : ns(1);
11   in din     : ns(8);
12   out ack    : ns(1)) {
13   iptype "spi_pl";
14   ipparm "wl=8";
15   ipparm "verbose=0";
16 }
```

Listing 4.3: SPI CPL ipblock

for SPI can be parameterized. The `ipparm ''wl=8''` sets the SPI to operate on 8-bit data. The CAL for the SPI has two parameters - data width and packet size. The `ipparm`

```
1  ipblock spi_al (
2      // cpl interface
3    in pvalid   : ns(1);
4    in pdin     : ns(8);
5    out pexists : ns(1);
6    out pdout   : ns(8);
7    in pack     : ns(1);
8      // cil interface
9    out ivalid  : ns(1);
10   out idout   : ns(8);
11   in iexists  : ns(1);
12   in idin     : ns(8);
13   out iack    : ns(1);
14   // fifo control signals
15   in rfull    : ns(1)) {
16   iptype "spi_al";
17   ipparm "wl=8";
18   ipparm "size=16";
19   ipparm "verbose=0";
20 }
```

Listing 4.4: SPI CAL ipblock

`''size=16''` statement sets the packet size to 16 for all transfers. The CAL on the other end of the channel abstraction (MCU) should also have the same packet size. Otherwise, the data transfers will not synchronize. It implements the FSM described in Figure 3.4.

4.2.2 UART

The following listings are the CAL and CPL implementations for UART. Like SPI, they too are parameterizable. The UART CPL supports baud rates from 9.6kbps to 250kbps and fully configurable byte frame. The `ipparm ''frame=8N1''` sets the byte frame to 8-bit data, no parity (N) and 1 stop bit. The data width can be 5, 6, 7 or 8. Replace 'N' with 'O' for generation and transmission of odd parity and 'E' for even parity. The GEZEL language

We have benchmarked our communication library to characterize each interface in terms of

```
1  ipblock uart_pl (
2    // uart ports
3    in rxd     : ns(1);
4    out txd    : ns(1);
5    // cal interface ports
6    out valid  : ns(1);
7    out dout   : ns(8);
8    in exists  : ns(1);
9    in din     : ns(8);
10   out ack    : ns(1)) {
11   iptype "uart_pl";
12   ipparm "fMHz=50";
13   ipparm "baud=115200";
14   ipparm "frame=8N1";
15   ipparm "verbose=0";
16 }
```

Listing 4.5: UART CPL ipblock

```
1  ipblock uart_al (
2      // cpl interface ports
3    in pvalid   : ns(1);
4    in pdin     : ns(8);
5    out pexists : ns(1);
6    out pdout   : ns(8);
7    in pack     : ns(1);
8      // cil interface ports
9    out ivalid  : ns(1);
10   out idout   : ns(8);
11   in iexists  : ns(1);
12   in idin     : ns(8);
13   out iack    : ns(1);
14   // fifo control signals
15   in rfull    : ns(1)) {
16   iptype "uart_al";
17   ipparm "wl=8";
18   ipparm "size=16";
19   ipparm "verbose=0";
20 }
```

Listing 4.6: UART CAL ipblock

throughput, code segment size and area in hardware. All such performance numbers are available in Appendix A.

Chapter 5

VHDL Code Generation from GEZEL Ipblocks

Hardware designs in GEZEL can be automatically converted to synthesizable VHDL code using the GEZEL code generator tool called `fdlvhd`. A brief description of the usage of the tool can be found in [17].

For ipblocks, `fdlvhd` generates a black box. We need to supply a VHDL code for the ipblock manually if it needs to be synthesized. This is problematic for two reasons: firstly, a GEZEL hardware description may contain multiple ipblocks which may need to be synthesized. Manually adding VHDL code for all such modules after the tool is run decreases productivity. Secondly, ipblocks support parameterization which worsens this problem. The VHDL code needs to be modified whenever an ipblock parameter is changed, so that it behaves as expected.

To solve these problems, we have extended the `fdlvhd` tool to support ipblocks. It allows the automatic generation of user supplied VHDL code for ipblocks taking parameterizations into consideration. We will discuss this mechanism in detail in this chapter.

5.1 The fdlvhd tool

For an ipblock, VHDL code generation support has to be added and the `fdlvhd` tool has to be rebuilt. It is implemented completely in C++. We will explain the code generation process with the example of a RAM ipblock shown in Listing 5.1.

```
1  ipblock R(
2        in en      : ns(1);
3        in we      : ns(1);
4        in addr    : ns(5);
5        in di      : ns(8);
6        out do     : ns(8) {
7     iptype "ram_b";
8     ipparm "size=32";
9     ipparm "wl=8";
10 }
```

Listing 5.1: Block RAM ipblock

This is an ipblock which models a block-RAM with 8-bit data and 32 locations. The size and wordlength are parameterized. Xilinx provides a specific way to model the block RAM in VHDL so that when it is synthesized, it will map to block RAM on the FPGA. Our goal is to be able to generate the Xilinx-specific VHDL code for the block-RAM ipblock parameterized in size and word length.

We assume that the user is familiar with the development of ipblock simulation model which can be found here [18]. The design of the VHDL code generation mechanism for an ipblock is very similar to the design of its simulation model. There are three steps to add VHDL code generation support explained below.

5.1.1 Step 1: Design the VHDL Code

The VHDL code for Xilinx block-RAM is shown in Listing 5.2. Note that the simulation model of the ipblock should match its VHDL code to maintain fidelity in the behavior of the

```
1  library ieee;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4  library work;
5  use work.std_logic_arithext.all;
6
7  -- datapath entity
8  entity block_ram is
9  port(
10     en          : in std_logic;
11     we          : in std_logic;
12     addr        : in std_logic_vector(4 downto 0);
13     di          : in std_logic_vector(7 downto 0);
14     do          : out std_logic_vector(7 downto 0);
15     CLK         : in std_logic
16 );
17 end block_ram;
18
19 architecture RTL of block_ram is
20 -- signal declaration
21 type ram_type is array(31 downto 0) of std_logic_vector(7 downto 0);
22 signal RAM        : ram_type;
23 signal reg_addr   : std_logic_vector(4 downto 0);
24
25 begin
26
27 process(CLK)
28 begin
29      if (CLK'event and CLK = '1') then
30           if (en = '1') then
31                if (we = '1') then
32                     RAM(to_integer(unsigned(addra))) <= di;
33                end if;
34                reg_addr <= addr;
35           end if;
36      end if;
37 end process;
38
39 do <= RAM(to_integer(unsigned(reg_addr)));
40
41 end RTL;
```

Listing 5.2: Block RAM VHDL code

hardware. Once the VHDL design for the hardware module is available, we proceed to Step 2.

5.1.2 Step 2: C++ Implementation

`Fdlvhd` already generates the VHDL skeleton with libraries and the `entity` declaration. The main RTL - lines 20 through 40 in the Listing 5.2 only need to be generated. For this, we create a C++ component class derived from an abstract class `avhdvc` shown in Listing 5.3. This abstract class provides interfaces for generating VHDL code for ipblock components.

```
1  class avhdvc {
2   public:
3     avhdvc();
4     virtual bool isValid();
5     ~avhdvc();
6     virtual void setInstName(string);
7     virtual void setparm(char *);
8     virtual bool hasRst();
9     virtual void dump(vhdout *vhdlout, string clkname);
10 };
```

Listing 5.3: avhdvc abstract class

For the sake of convenience, the code Listings for the component class for block-RAM are provided in Appendix B, Listings B.1 and B.2.

The code generation mechanism uses C++ factory design pattern. The component class requires a constructor accepting a string argument. This argument is matched with the ipblock type name, for example, "ram_b" in the case of block-RAM. If the name does not match, the `isValid()` method needs to return false. In this way, at runtime, an object of base class `avhdvc` is validated for a particular ipblock, when its type name matches.

The `setInstName(string)` method stores the name of the ipblock instance. The `setparm(char *)` function records all parameters of the ipblock specified using the `ipparm` statements. It is exactly identical to the `setparm(char *)` method implemented in the design of the simulation model, which can be found in [18]. As shown in Listing B.2, the RAM size is obtained in the variable `memSize` and the word length in the variable `dataWidth`.

When the ipblock skeleton is generated, a reset (RST) input signal is automatically added.

The `hasRst()` method is used to override this. If the ipblock simulation model already contains a reset input, or if the reset input is not required, this method should return true.

Finally, the `dump(vhdout *vhdlout, string clkname)` method is used to add the RTL logic for the VHDL code. The `vhdlout` argument provides the file output stream for the block-RAM VHDL code through its methods `vhdlout::put()` and `vhdlout::putline()`. The latter is used when a new line is printed. The `vhdlout::increaseindent()` and `vhdlout::decreaseindent()` are used to increase and decrease the indentation respectively.

All these methods are shown in the Listing B.2.

5.1.3 Step 3: Recompiling the Code Generator

The GEZEL code generation does not support dynamic linking of shared libraries. We need to add the development files to the GEZEL code generation source code and compile manually. The source code can be obtained by installing `gezel-sources`.

```
$ sudo apt-get install gezel-sources
```

Extract the contents of `gezel-cgen.tar.gz` from `/opt/gezel-sources` to a suitable location in your home directory. Let's say that the VHDL component class declaration and methods for block-RAM are in the files `vhdMemory.cxx` and `vhdMemory.h`. Place these files in `cgen/component`. Now, edit the file `vhdvcfactory.cxx` in `cgen/component` to include the new header:

```
#include "vhdMemory.cxx"
```

In the `avhdvc *generateVhdVC(char *tname)` method, add the line

```
GENERATE(vhdMemory, tname);
```

This statement adds the class name `vhdMemory` to the list of ipblocks for which manual code generation is supported.

The last step in the compilation process is to add the new sources in the compile order. Open `cgen/component/Makefile.am` file and make the following changes:

```
libipcomponent_la_SOURCES = vhdvcfactory.cxx vhdvcfactory.h \
               avhdvc.cxx avhdvc.h \
               vhdMemory.cxx vhdMemory.h

pkginclude_HEADERS = vhdvcfactory.h \
        avhdvc.h \
        vhdMemory.h
```

Now compile the source code from the `cgen` directory by executing the following lines:

```
$ ./bootstrap
$ ./configure --enable-vhdl --enable-igc
          --with-gezel=/opt/gezel --prefix=/opt/gezel
$ make
$ sudo make install
```

This will install `fdlvhd` in the default location `/opt/gezel/bin`, updated with automatic code generation support for block-RAMs. In Appendix C, we have provided a catalog of ipblocks for which the code generation support is available, pre-compiled in the latest version of GEZEL, `gezel-2.5.8`.

Chapter 6

Proof of Concept

In this chapter, we put everything from the previous chapters together to build an application: a 128-bit AES encryption algorithm mapped in hardware. We first show how it is co-designed in nesC and GEZEL using our layered communication architecture. We will use SPI and UART as our underlying interfaces and show that the effort required to switch between interfaces is very little. The simulation of the application using the SUNSHINE simulator follows the application development stage. Finally, we implement the design on our prototype sensor board which connects an MCU with an FPGA.

6.1 Application Design

The AES encryption algorithm is mapped entirely in hardware. Figure 6.1 shows the complete application design. The nesC code on the MCU prepares 16 bytes of text and 16 bytes of key. The text and the key packets travel through the layers into the FPGA. The AES co-processor is connected to the `send` and `receive` interfaces of CIL. It reads out the text and the key, performs encryption and writes back the encrypted result. The MCU then requests for the result back from the FPGA.

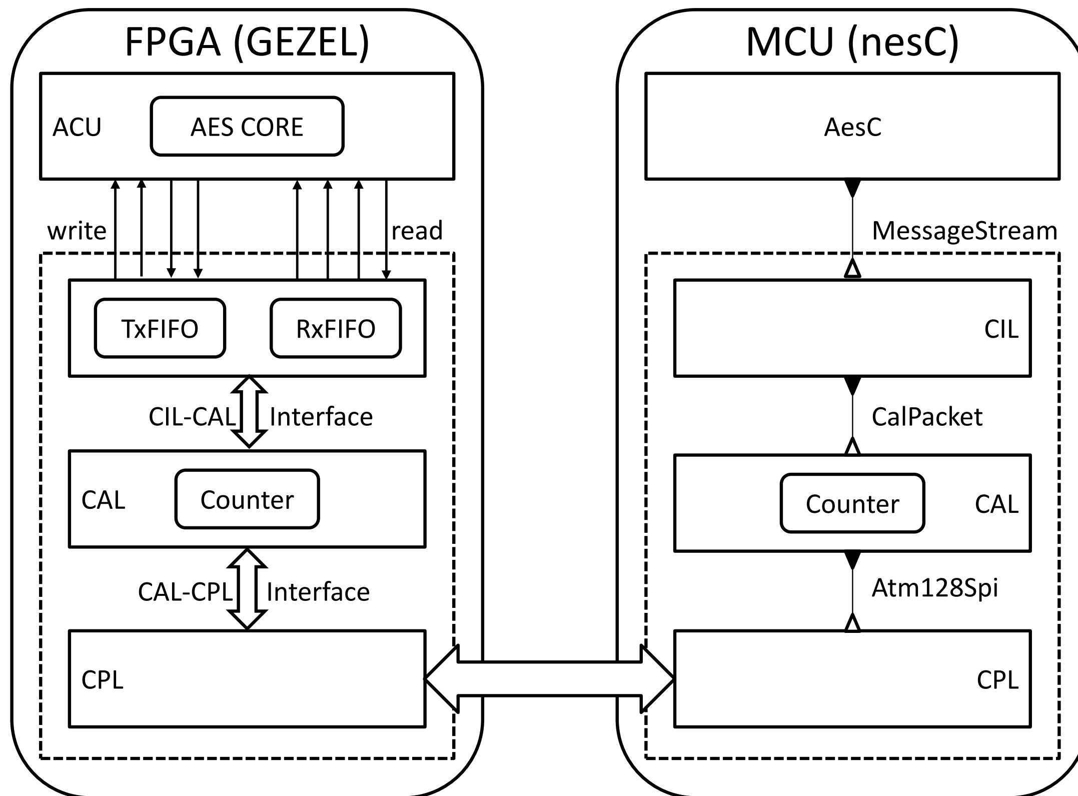

Figure 6.1: AES Application using SPI (schematic)

Listing 6.1 represents the main application component `AesC`. It *uses* the interface `MessageStream` provided by the component `ChannelC`. The complete task mapped in hardware, thus reduces to the 4 lines in Listing 6.1.

```
1  call MessageStream.init();                      // initialize
2  call MessageStream.send(key, 16);               // send key
3  call MessageStream.send(text_in, 16);           // send the text
4  call MessageStream.receive(text_out, 16); // get back the result
```

Listing 6.1: The AES application (AesC component)

The `ChannelC` component decides which underlying interface to use. As shown in the Listing 6.2, it actually wraps around the CIL layer of SPI called `CilSpiC`. Thus, all messages transferred using the `MessageStream` interface, in fact, use the SPI bus.

There are application specific settings which are to be put in a header file, say `AesC.h`. These

```
1 configuration ChannelC {
2   provides interface MessageStream;
3 }
4 implementation {
5   components CilSpiC as HWCHannel;
6   MessageStream = HWCHannel;
7 }
```

Listing 6.2: The ChannelC component

include the `PACKET_SIZE` and other communication interface specific settings discussed in the previous chapter.

```
const uint16_t    PACKET_SIZE     = 16;
```

The hardware side of the implementation is fairly straightforward. The ACU module implements an FSM to read from and write to the FIFOs. It reads 16 bytes of key followed by 16 bytes of text from the input FIFO. As soon as the text and key are ready, an AES encryption module instantiated within the ACU is signaled to start the encryption. When the encryption is done, the ACU writes 16 bytes of encrypted text to the output FIFO.

The packet size has to maintained the same in hardware too. This is done by using the `ipparm ''size=16''` statement in the `spi_al` ipblock (Listing 4.4) and `ipparm ''depth=16''` statement in the FIFO ipblocks used within the CIL component.

6.2 Simulation Results

We ran the AES application in software as well as hardware and obtained a cycle count in both cases. We used the GEZEL cycle accurate simulator to obtain the performance numbers.

Table 6.1 provides the performance numbers for the AES encryption task in comparison to a reference software implementation, obtained from the simulation stage. Despite the serial

Table 6.1: AES computational costs

Implementation	Cycles	Speedup
MCU (reference software)	19364	1
FPGA (w/o comm overhead)	10	1936.4
FPGA (with comm overhead)	5804	**3.3**

communication overhead, we observe that the AES encryption algorithm as a task ported in hardware performs 3.3 times faster than software.

6.3 Prototype

After simulation, we map the application in our prototype platform. Our prototype platform consists of a BDMicro board with AVR ATMega128 MCU and a Spartan-3E FPGA board. We use the same nesC binary used in our previous simulation step to configure the AVR MCU. The open source software `avrdude` [19] is used to download the binary onto the BDMicro board. Likewise, the `fdlvhd` tool generates VHDL code for all modules in GEZEL, including the communication layer ipblocks. We use Xilinx ISE to generate the bitstream for the Spartan 3E FPGA. With the help of the LCD display, we verify that the co-design works correctly.

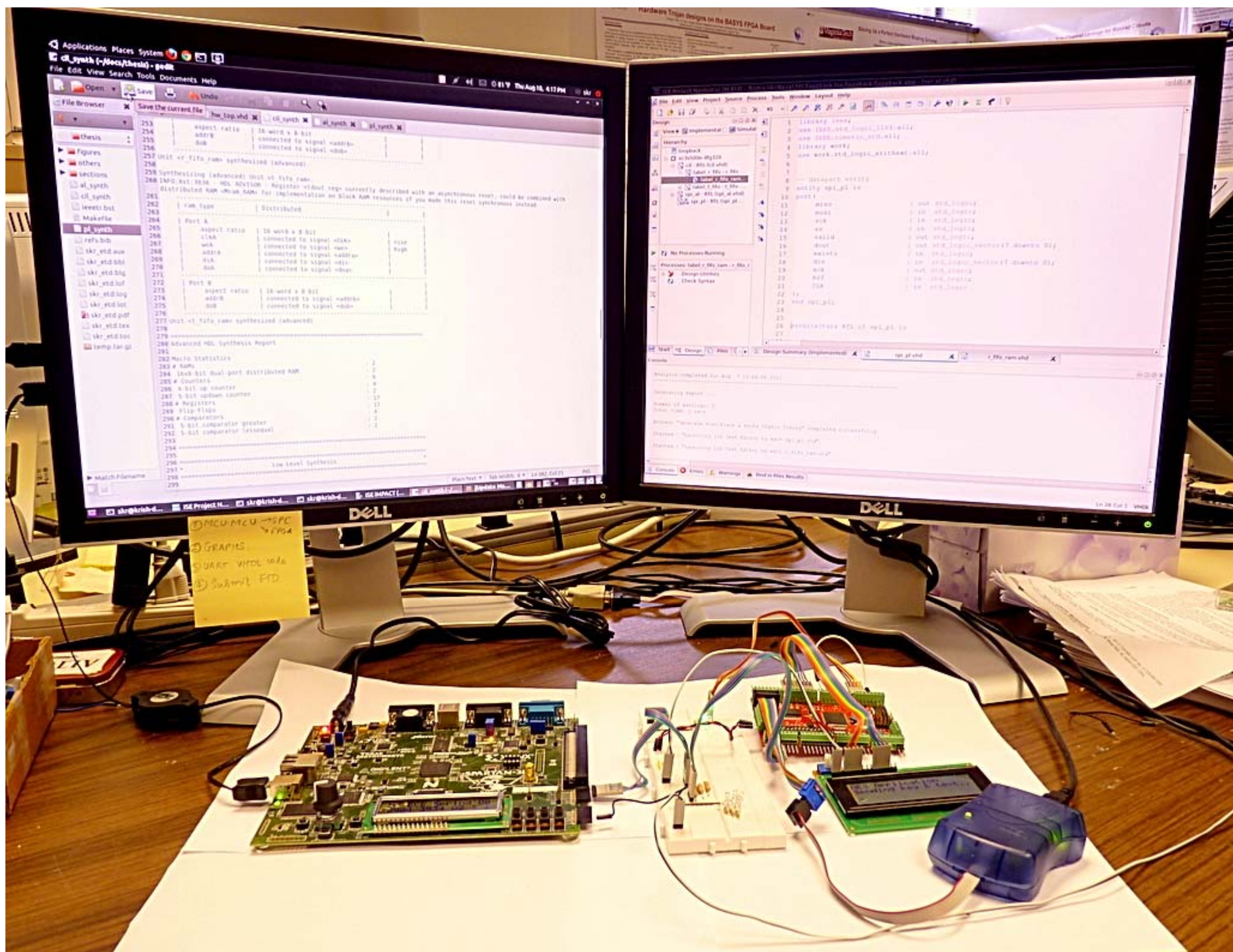

Figure 6.2: Sensor node prototype with AVR and FPGA boards

Chapter 7

Conclusion and Future Work

We presented a three-layered abstraction model to integrate hardware and software on constrained sensor node platforms. In contrast with the existing sensor node development environments, we support a flexible sensor node topology thanks to the use of hardware modeling. We are currently extending our communication library to support other MCU interfaces and we plan to develop a sensor node board for our design environment which will pair an MCU with a low-power FPGA. Our future work will focus on performing an accurate assessment of energy savings on our sensor board.

Bibliography

[1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *SIGPLAN Not.*, vol. 35, pp. 93–104, November 2000. [Online]. Available: http://doi.acm.org/10.1145/356989.356998

[2] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, ser. LCN '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462. [Online]. Available: http://dx.doi.org/10.1109/LCN.2004.38

[3] A. Boulis, C.-C. Han, and M. B. Srivastava, "Design and implementation of a framework for efficient and programmable sensor networks," in *Proceedings of the 1st international conference on Mobile systems, applications and services*, ser. MobiSys '03. New York, NY, USA: ACM, 2003, pp. 187–200. [Online]. Available: http://doi.acm.org/10.1145/1066116.1066121

[4] J. Rabaey, *Low Power Design Essentials*. Springer, 2009.

[5] J. Ko, C. Lu, M. Srivastava, J. Stankovic, A. Terzis, and M. Welsh, "Wireless sensor networks for healthcare," *Proceedings of the IEEE*, vol. 98, no. 11, pp. 1947 –1960, nov. 2010.

[6] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM Trans. Embed. Comput. Syst.*, vol. 3, pp. 461–491, August 2004. [Online]. Available: http://doi.acm.org/10.1145/1015047.1015049

[7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," *SIGPLAN Not.*, vol. 38, pp. 1–11, May 2003. [Online]. Available: http://doi.acm.org/10.1145/780822.781133

[8] P. Schaumont, *A Practical Introduction to Hardware-Software Codesign*. Springer, 2010.

[9] "Simulavr: an avr simulator," Website, `http://www.nongnu.org/simulavr/`.

[10] P. Levis, N. Lee, M. Welsh, and D. Culler, "Tossim: accurate and scalable simulation of entire tinyos applications," in *Computer Communications and Networks, International Conference on Embedded networked sensor systems*, 2003, pp. 126–137.

[11] J. Zhang, Y. Tang, S. Hirve, P. Schaumont, and Y. Yang, "A software-hardware emulator for sensor networks," in *Proc. 8th IEEE Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, 2011.

[12] V. Schaumont and I. Verbauwhede, "A component-based design environment for esl design," *Design Test of Computers, IEEE*, vol. 23, no. 5, pp. 338 –347, may 2006.

[13] M. Knezzevic, K. Sakiyama, Y. Lee, and I. Verbauwhede, "On the high-throughput implementation of ripemd-160 hash algorithm," in *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, july 2008, pp. 85 –90.

[14] B. Köpf and D. Basin, "An information-theoretic model for adaptive side-channel attacks," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 286–296. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315282

[15] J. Zhang, S. Iyer, S. Hirve, Y. Yang, and P. Schaumont, "Sunshine: A software-hardware emulator for sensor networks," Website, `http://rijndael.ece.vt.edu/sunshine/index.html`.

[16] V. Handziski, J. Polastre, J.-H. Hauer, C. Sharp, A. Wolisz, and D. Culler, "Flexible hardware abstraction for wireless sensor networks," in *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, jan.-2 feb. 2005, pp. 145 – 157.

[17] P. Schaumont, "Gezel code-generation tool," Website, `http://rijndael.ece.vt.edu/gezel2/codegeneration.html`.

[18] "Gezel library modules," Website, http://rijndael.ece.vt.edu/gezel2/library.html.

[19] "Avr downloader/uploader," Website, `http://savannah.nongnu.org/projects/avrdude/`.

[20] "Spartan-3e fpga datasheet," URL, `http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf`.

Appendix A

Communication Library Benchmarks

A.1 Performance Evaluation

We measure the performance of these communication interfaces using the following metrics - throughput measured in terms of number of cycles per byte and CPU utilization factor. We also calculate the code segment size to compare how large each implementation is. First, we willl describe the platform used for generating the benchmarks.

A.1.1 Benchmarking Platform

We use the simulation framework described in chapter 2 - SimulAVR integrated into GEZEL. We use a two MCU platform, one as the Master and the other, Slave. In other words, the ACU that we attach to the primary MCU is in fact a slave MCU. For each of the above interfaces, we measure the observed throughput. Figure A.1 illustrates the benchmarking platform.

The MCU sends a packet of data to the ACU. The packet length is set to 1024 bytes. The MCU retrieves the same packet back from the ACU, marking the completion of one round

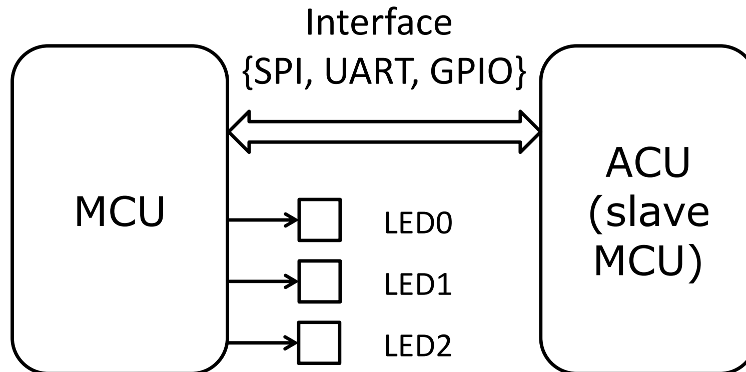

Figure A.1: The platform used for generating benchmarks

```
1  Initialize Interface;              // interface initialization
2  Initialize array: txData, rxData; // array of 1K bytes
3
4  LEDS := 1;                         // capture T0
5  Interface.Transmit(txData);        // Transmit 1K bytes
6
7  LEDS := 2;                         // capture T1
8  Interface.Recieve(rxData);         // receive 1K bytes
9
10 LEDS := 3;                         // capture T2
11 LEDS := match(txData, rxData);     // LEDs on if match
```

Listing A.1: Pseudo-code for benchmarks

trip. Leds attached to the MCU are toggled between the send and receive events. In the GEZEL description, the toggling events are captured and thus we obtain an accurate cycle count.

The listing above shows a very simple pseudo-code for the benchmark.

A.1.2 Throughput (cycles per byte)

Based on the pseudo-code above, T1 - T0 gives the **total cycles** for transmission and T2 - T1 gives the **total cycles** for reception. This includes to overhead of calculating the address of next byte to be transmitted and traversal of the data through the three layers of our

architecture. The **effective cycles per byte** is calculated by dividing the total cycle count for sending as well as receiving by 1024.

Some time is spent in waiting for the ongiong transfer to complete. In interrupt driven mode, the CPU waits for an interrupt and is essentially idle. In polled mode, the CPU constantly checks the status of the ongoing transfer. Whatever the case be, we recognize the wait time as an important parameter. It is the same as **absolute cycles per byte** in the case of serial interfaces. It can be calculated theoretically when the serial transfer rate is known.

A.1.3 CPU Load

The CPU is effectively stalled during an ongoing transfer. It has to either wait for an interrupt or constantly poll the status of the transfer in progress. This is the **wait time** within the **total cycle** count. The remaining time is spent in executing the code pertaining to the layers of the interface architecture. This is the **total time without polling** (calculated per byte). If the **total time without polling** is less, it means the wait time is more, or the CPU is idle, longer. To characterize the interfaces, we define the **CPU load** as the ratio of **total time without polling** over the **effective cycles** per byte.

SPI

The following are some figures depicting the performance measure for SPI.

		SENDING / RECEIVING			
A	B	C	D	E	F
SCK frequency Fosc / f	Absolute cycles per byte = 8 * f	Total cycles (1KB)	Total cycles without polling (1KB)	Effective cycles per byte = C / 1024	Load (%) = D / C
Fosc / 2	16	38924	22540	38	57.9
Fosc / 4	32	54284	21516	53	39.6
Fosc / 8	64	88076	22540	86	25.6
Fosc / 16	128	152588	21516	149	14.1
Fosc / 32	256	284684	22540	278	7.9
Fosc / 64	512	545804	21516	533	3.9
Fosc / 128	1024	1071116	22540	1046	2.1

Figure A.2: Performance measure: polled mode

		SENDING / RECEIVING			
A	B	C	D	E	F
SCK frequency Fosc / f	Absolute cycles per byte = 8 * f	Total cycles (1KB)	Total cycles without polling (1KB)	Effective cycles per byte = C / 1024	Load (%) = D / C
Fosc / 2	16	146444	130060	143	88.8
Fosc / 4	32	166924	134156	163	80.4
Fosc / 8	64	197644	132108	193	66.8
Fosc / 16	128	264204	133132	258	50.4
Fosc / 32	256	392204	130060	383	33.2
Fosc / 64	512	658444	134156	643	20.4
Fosc / 128	1024	1180684	132108	1153	11.2

Figure A.3: Performance measure: interrupt-driven mode

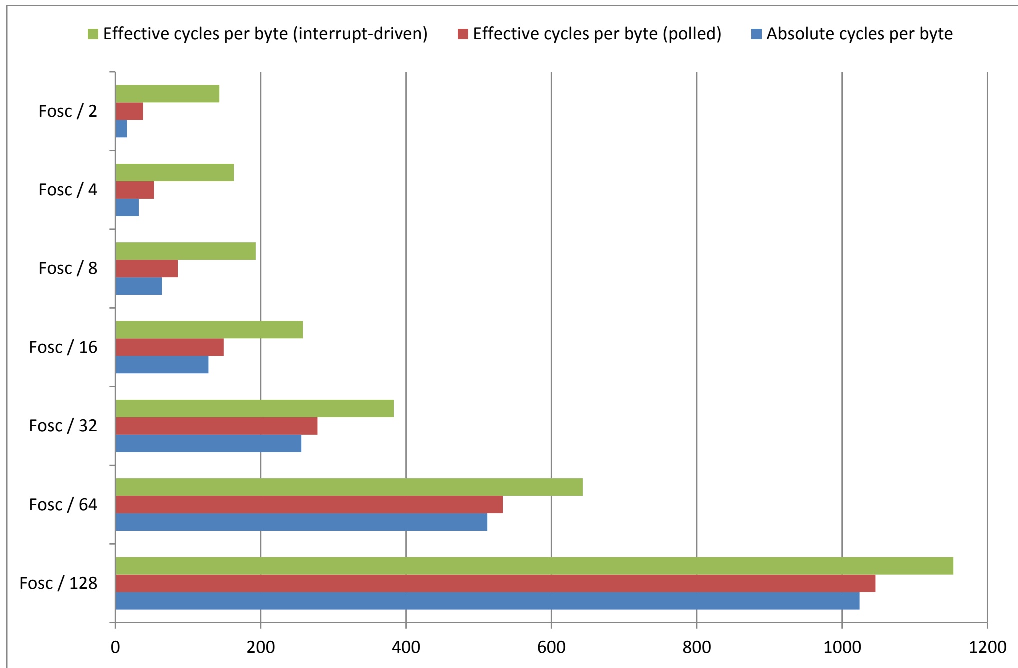

Figure A.4: SPI: Graph showing absolute and effective cycles per byte

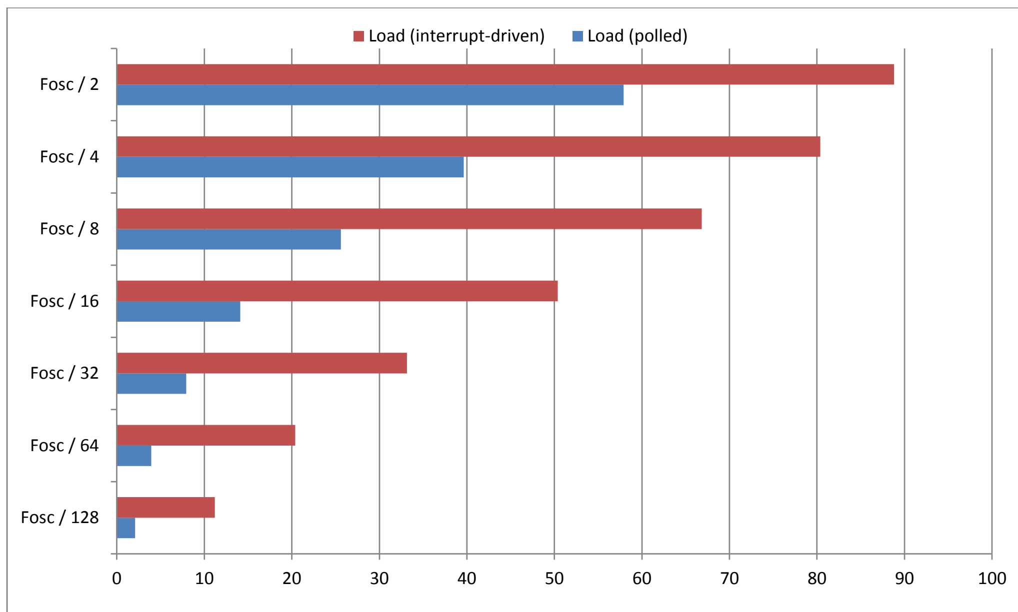

Figure A.5: SPI: Graph depicting the CPU load (polled and interrupt-driven)

		SENDING				RECEIVING			
A	B	C	D	E	F	G	H	I	J
Baud rate	Absolute cycles per byte	Total cycles (1KB)	Total cycles without polling (1KB)	Effective cycles per byte = C / 1024	Load (%) = D / C	Total cycles (1KB)	Total cycles without polling (1KB)	Effective cycles per byte = G / 1024	Load (%) = H / G
500000	320	327045	37903	319	11.6	328386	30740	321	9.4
250000	640	654072	37903	639	5.8	656706	30740	641	4.7
230400	640	654072	37903	639	5.8	656706	30740	641	4.7
115200	1280	1308186	37903	1278	2.9	1313412	30740	1283	2.3
76800	2080	2125611	37903	2076	1.8	2134290	30740	2084	1.4
57600	2720	2779626	37903	2715	1.4	2790996	30740	2696	1.1
38400	4160	4251201	37903	4152	0.9	4268580	30740	4169	0.7
28800	5440	5559294	37903	5429	0.7	5582009	30740	5451	0.6

Figure A.6: Performance measure: polled mode

UART

The tables A.6 and A.7 provide the number of cycles for UART at baud rates ranging from 28.8k to 500k. The Figures A.8 and A.9 are the graphs which correspond to the measured throughput and the load on the CPU.

		SENDING				RECEIVING			
A	B	C	D	E	F	G	H	I	J
Baud rate	Absolute cycles per byte	Total cycles (1KB)	Total cycles without polling (1KB)	Effective cycles per byte = C / 1024	Load (%) = D / C	Total cycles (1KB)	Total cycles without polling (1KB)	Effective cycles per byte = G / 1024	Load (%) = H / G
500000	320	327581	198678	320	60.7	328536	204822	321	62.3
250000	640	654879	198678	640	30.3	656559	204822	641	31.2
230400	640	654879	198678	640	30.3	656559	204822	641	31.2
115200	1280	1309603	198678	1279	15.2	1312674	204822	1282	15.6
76800	2080	2128011	198678	2078	9.3	2132728	204822	2083	9.6
57600	2720	2782551	198678	2717	7.1	2788878	204822	2724	7.3
38400	4160	4255577	198678	4156	4.7	4264690	204822	4165	4.8
28800	5440	5565013	198678	5434	3.6	5577094	204822	5446	3.7

Figure A.7: Performance measure: interrupt-driven mode

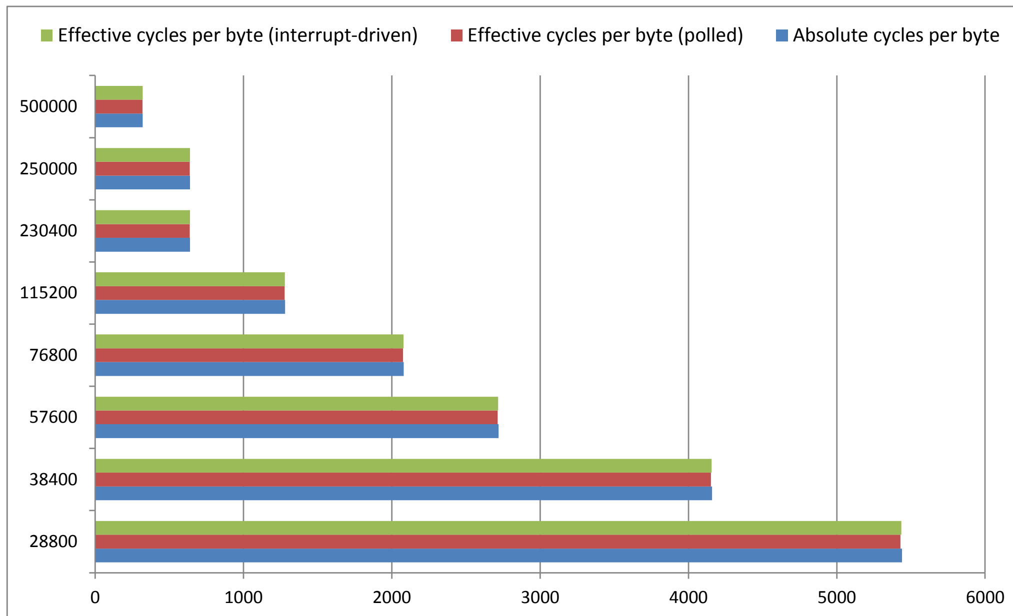

Figure A.8: UART: Graph showing absolute and effective cycles per byte

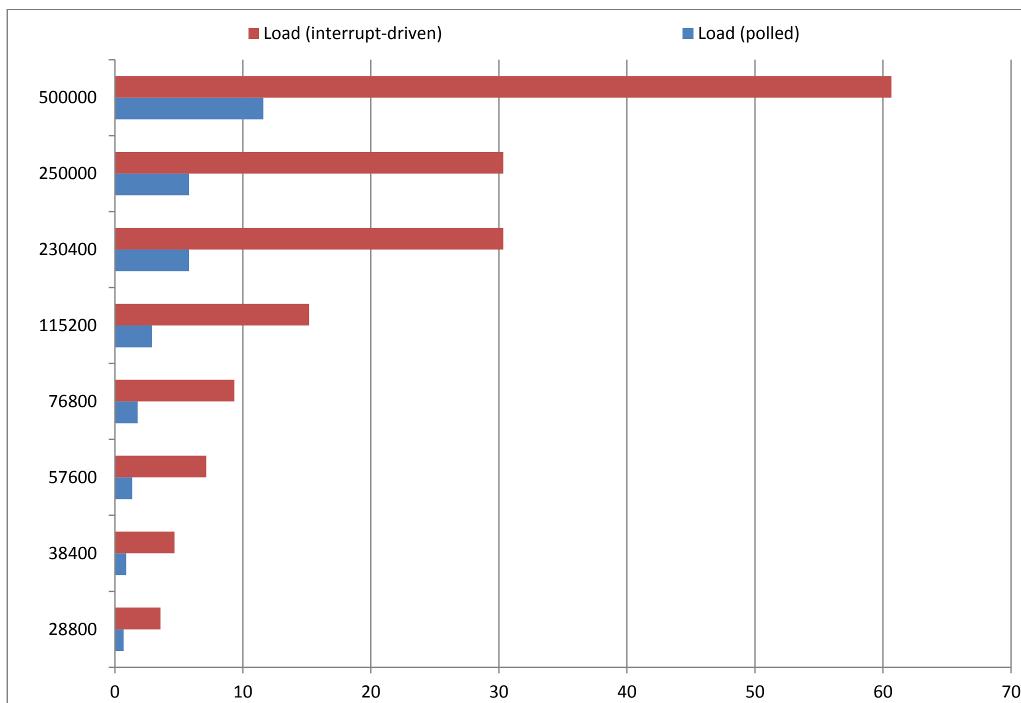

Figure A.9: UART: Graph depicting the CPU load (polled and interrupt-driven)

		SENDING / RECEIVING			
A	B	C	D	E	F
# of bits transferred at a time	Absolute cycles per byte	Total cycles (1KB)	Total cycles without polling (1KB)	Effective cycles per byte = C / 1024	Load (%) = D / C
1	8	700479	878733	684	71.6
4	2	112648	71688	110	63.6
8	1	64512	44032	63	68.3
16	1	37384	16904	37	45.2

Figure A.10: Performance measure: polled mode

GPIO

The following are some figures depicting the performance measure for GPIO interface.

		SENDING / RECEIVING			
A	B	C	D	E	F
# of bits transferred at a time	Absolute cycles per byte	Total cycles (1KB)	Total cycles without polling (1KB)	Effective cycles per byte = C / 1024	Load (%) = D / C
1	8	1226893	1169549	1199	95.3
4	2	218282	203946	213	93.4
8	1	107642	100474	105	93.3
16	1	60295	53127	59	88.1

Figure A.11: Performance measure: interrupt-driven mode

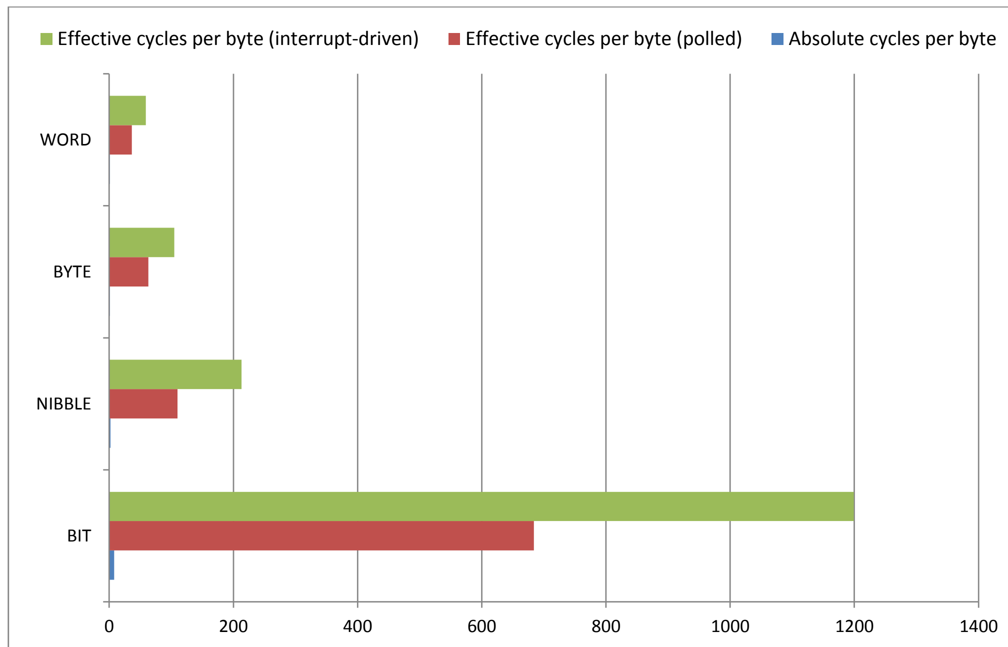

Figure A.12: GPIO: Graph showing absolute and effective cycles per byte

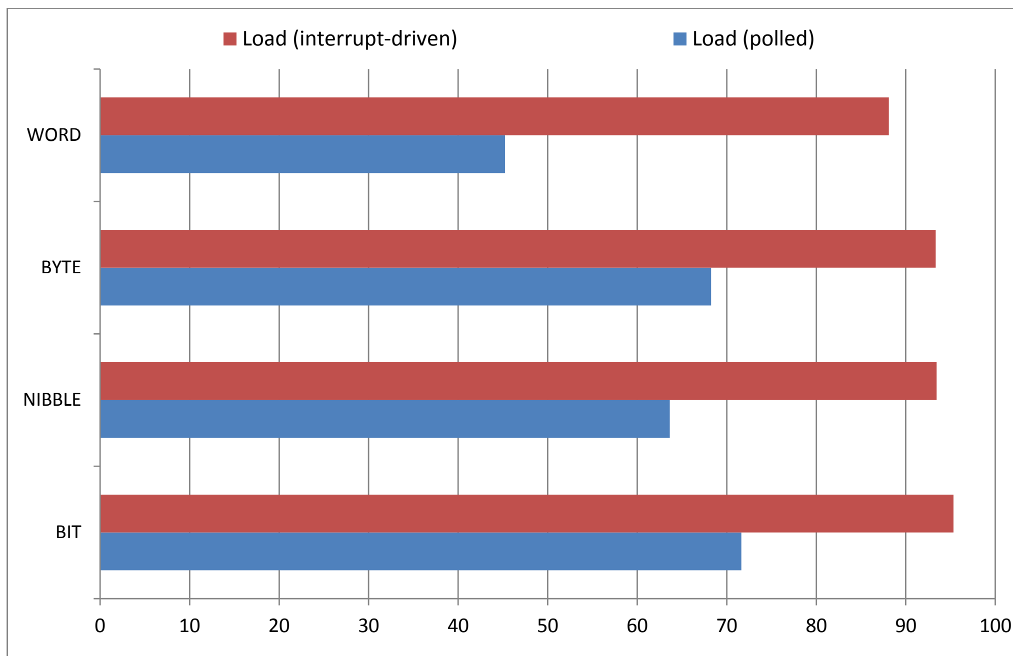

Figure A.13: GPIO: Graph depicting the CPU load (polled and interrupt-driven)

A.2 Hardware Resource Utilization

The SPI and UART layers in hardware are designed as ipblocks with code generation support. The VHDL code for the three layers are synthesized on a Spartan 3E FPGA (device XC3S500E) board using Xilinx ISE 12.4. We provide the implementation details of the communication layers for UART and SPI interfaces in terms of slices, LUTs and flip flops. More information about the Spartan 3E FPGA can be found here [20].

A.2.1 SPI

The device utilization summary for SPI interface for the CIL, CAL and CPL is as follows:

```
Selected Device : 3s500efg320-4

 Number of Slices:                      91  out of   4656     1%
 Number of Slice Flip Flops:            68  out of   9312     0%
 Number of 4 input LUTs:                177  out of   9312    2%
    Number used as logic:               145
    Number used as RAMs:                32
```

The CIL instantiates two FIFOs mapped in LUTs, each having a depth of 16. This is indicated in the summary above (32 LUTs used as RAMs). The RAM depth of the FIFOs within CIL is an application specific setting and should match the packet size parameter of CAL. Alternately, the FIFOs could be mapped in the dedicated block RAMs which will free up the 32 LUTs.

The resource usage as indicated above is very minimal - the main ACU core which will attach to the CIL layer has 99% of the available resources.

A.2.2 UART

The device utilization summary for the UART interface which includes the three layers - CIL, CAL and CPL, is as follows:

```
Selected Device : 3s500efg320-4

 Number of Slices:                      98  out of   4656     2%
 Number of Slice Flip Flops:            78  out of   9312     1%
 Number of 4 input LUTs:                188  out of   9312    2%
    Number used as logic:               156
    Number used as RAMs:                32
```

Compared to the SPI interface, UART has a slightly larger resource utilization due to the complex nature of the CPL implementation. Unlike SPI, the UART interface needs to generate an internal clock at the specified baud rate for transmitting bits and also, sample the RxD pin for reception. The resource usage is still, minimal, with only 2% of the slices used.

Appendix B

VHDL Code Generation Source File for Block-RAM

The following Listings B.1 and B.2 are complete source files for VHDL code generation support for a block-RAM ipblock.

```
1  #include "avhdvc.h"
2
3  class vhdMemory : public avhdvc {
4   private:
5    string instanceName;
6    int memSize;
7    int dataWidth;
8    int addressWidth;
9    bool valid;
10
11  public:
12   vhdMemory(string);
13   ~vhdMemory();
14   void setInstName(string);
15   void setparm(char *parmname);
16   bool isValid();
17   void setPortMap( map <symid, vhdvar*> *plist );
18   void dump(vhdout *vhdlout, string clkname);
19   bool hasRst();
20 };
```

Listing B.1: Block-RAM component class declaration (vhdMemory.h)

```
1  #include "vhdMemory.h"
2  #include "matchparam.h"
3  #include "vhdout.h"
4  #include "vhdvdp.h"
5  #include "vhdvar.h"
6  #include <fstream>
7
8  vhdMemory::vhdMemory(string name) {
9    if (!name.compare("ram_b"))
10     valid = true;
11   else
12     valid = false;
13 }
14
15 bool vhdMemory::isValid() {
16   return valid;
17 }
18
19 void vhdMemory::setInstName(string inName) {
20   instanceName = inName;
21 }
22
23 bool vhdMemory::hasRst() {
24   return true;
25 }
26
27 void vhdMemory::setparm(char * parmname) {
28   gval *v = make_gval(32, 0);
29   if (matchparm(parmname, "size", *v)) {
30   memSize = v->toulong();
31   } else if (matchparm(parmname, "wl", *v)) {
32   dataWidth = v->toulong();
33   } else
34     cout << instanceName << ": Parameter not recognized. \n";
35
36   // calculate addressWidth = ceiling[log2(Memsize)]
37   addressWidth = 0;
38   unsigned long x = 0x80000000;
39   do {
40     if (x & memSize) {
41       while (x >>= 1)
42         addressWidth++;
43       break;
44     }
45   } while (x >>= 1);
46 }
```

```
47 void vhdMemory::dump(vhdout *vhdlout, string clkname) {
48   vhdlout->comment("signal declaration");
49   vhdlout->putline() << "type ram_type is array (" << memSize - 1;
50   vhdlout->put() << " downto 0) of std_logic_vector (" << dataWidth - 1;
51   vhdlout->put() << " downto 0);" << "\n";
52   vhdlout->putline() << "signal RAM : ram_type;" << "\n";
53   if (memSize > 2) {
54     vhdlout->putline() << "signal reg_addr : std_logic_vector"<< addrwl - 1;
55     vhdlout->put() <<" downto 0);" << "\n";
56   } else
57     vhdlout->putline() << "signal reg_addr : std_logic;" << "\n";
58   vhdlout->putline() << "begin" << "\n";
59   vhdlout->increaseindent();
60   vhdlout->putnewline() << "process(" << clkname << ")" << "\n";
61   vhdlout->putline() << "begin" << "\n";
62   vhdlout->increaseindent();
63   vhdlout->putline() << "if (" << clkname << "\'event and ";
64   vhdlout->put() << clkname << " = \'1\') then " << "\n";
65   vhdlout->increaseindent();
66   vhdlout->putline() << "if (en = \'1\') then" << "\n";
67   vhdlout->increaseindent();
68   vhdlout->putline() << "if (we = \'1\') then" << "\n";
69   vhdlout->increaseindent();
70   vhdlout->putline() << "RAM(to_integer(unsigned(addr))) <= di;" << "\n";
71   vhdlout->decreaseindent();
72   vhdlout->putline() << "end if;" << "\n";
73   vhdlout->putline() << "reg_addr <= addr;"<<"\n";
74   vhdlout->decreaseindent();
75   vhdlout->putline() << "end if;" << "\n";
76   vhdlout->decreaseindent();
77   vhdlout->putline() << "end if;" << "\n";
78   vhdlout->decreaseindent();
79   vhdlout->doindent();
80   vhdlout->putline() << "end process; " << "\n";
81   vhdlout->putline() << "do <= RAM(to_integer(unsigned(reg_addr)));" << "\n";
82   vhdlout->decreaseindent();
83 }
```

Listing B.2: Block-RAM component class methods (vhdMemory.cxx)

Appendix C

Catalog of Ipblocks with Code Generation Support

The following is a catalog of ipblocks with pre-complied code generation support, developed using the methodology in Chapter 5. The generated VHDL code behaves exactly like the GEZEL simulation model. The keyword `ipblock` is used to characterize the hardware model as a black-box, and the keyword `iptype` indicated the hardware model it represents. A list of parameters for the ipblock are specified through the `ipparm` statements. For the following ipblocks, the essential parameters are shown along with the optional ones in square brackets. Optional I/O ports for the ipblock are also shown in square brackets.

1. Single/Dual port Block/Distributed RAM

   This ipblock models a Xilinx compatible RAM module either mapped in the dedicated RAM blocks or in the LUTs of the FPGA. The dual-port RAM versions have two read ports and one write port. The size of the RAM and the data word-length are parameterized. The ipblocks are shown in Listings C.1, C.2, C.3, and C.4.

```
1  ipblock R(
2        in en      : ns(1);
3        in we      : ns(1);
4        in addr    : ns(log2(len));
5        in di      : ns(log2(width));
6        out do     : ns(log2(width)) {
7   iptype "ram_b";
8   ipparm "size=len";
9   ipparm "wl=width";
10  [ipparm "file=name";]
11  [ipparm "base=num";]
12  [ipparm "verbose=1";]
13 }
```

Listing C.1: Single-port block RAM

```
1  ipblock R(
2      in we      : ns(1);
3      in addr    : ns(log2(len));
4      in di      : ns(log2(width));
5      out do     : ns(log2(width)) {
6   iptype "ram_d";
7   ipparm "size=len";
8   ipparm "wl=width";
9   [ipparm "verbose=1";]
10 }
```

Listing C.2: Single-port distributed RAM

```
1  ipblock R(
2      in en      : ns(1);
3      in we      : ns(1);
4      in addra   : ns(log2(len));
5      in addrb   : ns(log2(len));
6      in di      : ns(log2(width));
7      out doa    : ns(log2(width));
8      out dob    : ns(log2(width)) {
9   iptype "dpram_b";
10  ipparm "size=len";
11  ipparm "wl=width";
12  [ipparm "verbose=1";]
13 }
```

Listing C.3: Dual-port block RAM

```
1  ipblock R(
2      in we      : ns(1);
3      in addra   : ns(log2(len));
4      in addrb   : ns(log2(len));
5      in di      : ns(log2(width));
6      out doa    : ns(log2(width));
7      out dob    : ns(log2(width)) {
8   iptype "dpram_d";
9   ipparm "size=len";
10  ipparm "wl=width";
11  [ipparm "verbose=1";]
12 }
```

Listing C.4: Dual-port distributed RAM

2. Shift register

```
1  ipblock R(
2    in si      : ns(len);
3    out so     : ns(len);
4    [in cen    : ns(1);]       // optional clock enable input
5    [in set    : ns(1);]       // optional active high synchronous set
6    [in rst    : ns(1);] {     // optional active high asynchronous reset
7                               // note that in VHDL, this is implicitly added
8                               // as RST
9   iptype "shiftreg";
10  ipparm "wl=len";
11  ipparm "depth=m";
12  [ipparm "verbose=1";]
13 }
```

Listing C.5: Shift register

As the name suggests, this models a shift register mapped on LUTs. Again, the word-length and the depth of the shift register module is parameterized. It is shown in Listing C.5.

3. FIFOs

```
1  ipblock R(
2    in wr_en       : ns(1);
3    in din         : ns(len);
4    in rd_en       : ns(1);
5    out dout       : ns(len);
6    out full       : ns(1);
7    out empty      : ns(1);
8    [in rst        : ns(1)]) {
9   iptype "fifo";
10  ipparm "wl=len";
11  ipparm "depth=m";
12  ipparm "type=distributed";    // or block
13  ipparm "mode=fwft";           // std and first word fall through
14  [ipparm "verbose=1";]
15 }
```

Listing C.6: FIFO

The FIFO ipblock shown in Listing C.6 has four essential parameters - The word length, FIFO depth, type and the mode. For the FIFO to operate predictably, the depth has to be a power of 2. The 'type' parameter specifies whether the FIFO is to be mapped in the dedicated block RAM module or in the LUTs. The 'mode' parameter is used to model a special FIFO feature - "First Word Fall Through" (fwft). In this mode, as

soon as the first word is written to the FIFO, it is available in the output line, even if the FIFO read strobe is not active. Thus the FIFO reads are advanced by a clock cycle. In the 'std' mode, the data available on the output line is available only a clock cycle later.

4. SPI CAL layer

   The CAL layer of the SPI interface is designed as an ipblock. Hence, code generation for the SPI CAL layer is also provided. The automatically generated VHDL code has been verified using ModelSim. Please refer to section 4.2.1 (Listing 4.4) for a brief description.

5. SPI CPL layer

   The CPL layer of the SPI is also modeled as an ipblock, shown in Listing 4.3.

6. UART CAL layer

   The CAL layer of UART is described in section 4.2.2, Listing 4.6.

7. UART CPL layer

   The CPL layer of UART is described in section 4.2.2, Listing 4.5.