The Design of an SoC-based Programmable Controller Development Platform

Alexandre Courtemanche

Department of Electrical & Computer Engineering
McGill University
Montreal, Canada

December 2014

A thesis submitted to McGill University in partial fulfilment of the requirements for the degree of Master of Engineering.

2014/12/02

Abstract

This thesis presents the development of the basis for an industrial controller based on System-on-Chip FPGA (SoC FPGA) technology as well as its accompanying suite of development tools. The objective of the project is to design an industrial controller development platform that can gracefully handle both high-level and low-level functionalities. A demonstration of Hardware-in-the-Loop (HIL) simulation with a graphical interface on a SoC FPGA using open-source software is one of the main pillars of the contributions of this thesis. First, a review of the embedded system design methodologies, the basics of co-processor design, and of development environments is presented. Next, the controller architecture's design process, which has produced multiple prototypes, is shown. The first prototype uses a decoupled architecture with a separate Central Processing Unit (CPU) and Field Programmable Gate Array (FPGA). A PCB demonstrating memory accesses from a microcontroller has been designed. Another prototype simulates a decoupled architecture composed of a powerful ARM core and an FPGA connected by PCI-Express (PCI-e). The most recent design is one based on SoC FPGA technology. To show the possibilities of this platform, a suite of example digital IP cores have been designed and simulated. Furthermore, a collection of development tools has been assembled and configured to enable developers to use this platform. In addition to the standard GNU tools, the thesis puts an emphasis on the modification of open-source simulation software to enable development with HIL simulation.

Sommaire

Ce mémoire de maîtrise présente l'élaboration de la base d'un contrôleur industriel, qui sera construit à partir de la technologie des SoC FPGAs, ainsi que la collection d'outils de développement qui l'accompagne. L'objectif du projet est de concevoir un plateforme de développement pour des contrleurs industriels qui possèdent à la fois des fonctionalités de bas-niveau et de haut-niveau. La démonstration d'une simulation HIL avec interface graphique qui utilise des logiciels libres est l'une des contributions académique principales de ce mémoire. Premièrement, une révision des méthodologies en conception de systèmes embarqués, de la base de conception de co-processeurs, ainsi que des environnements de développement est présenté. Ensuite, le procédé de développement de l'architecture, qui a produit plusieurs prototypes, est démontré. Le premier prototype emploie une architecture découplée qui utilise un processeur et FPGA séparé. Un circuit qui démontre les accès mémoire á partir d'un microcontrôleur a été fabriqué. Un autre prototype simule une architecture découplée qui est composée d'un processeur ARM et d'un FPGA connecté par PCI-e. La plus récente conception est basée sur la technologie SoC FPGA. Pour faire une démonstration des possibilités de cette plateforme, une collection de modules numériques ont été conçus et simulés. De plus, une collection d'outils de développement a été assemblée et configurée afin permettre aux développeurs d'utiliser cette plateforme. En plus des logiciels GNU standards, ce mémoire de maîtrise mets l'accent sur la modification des logiciels libres de simulation afin de permettre le développement avec de la simulation HIL.

Acknowledgments

First, I would like to thank my parents André Courtemanche and Ingrid Pitchen for their support and dedication to my education. It is thanks to their encouragement during the hard times and their constant determination to seeing me succeed that it has been possible for me to attain one of the most difficult achievements in higher education. I consider this thesis the pinnacle achievement of my academic career and it would not have been possible without them.

Secondly, I would like to thank my supervisor Zeljko Zilic for giving me the opportunity to work in a world-class academic research laboratory and to learn from some of the top researchers in electrical engineering. I will also give special thanks to Jean-Samuel Chenard for the generous donation of his time and expertise as well as the opportunity to collaborate with his company *Motsai Research*. From explaining the subtleties of applying for government scholarships, to reviewing the layout of my first PCB, and meticulously reviewing this thesis, Jean-Samuel has helped me almost every step of the way. I want to thank my friend and colleague Ben Nahill for helping me solve some of the most difficult engineering problems and for teaching me a great deal about PCB design. His hard work, determination, and 'can-do' attitude has inspired me to become a better engineer. I wish to thank all of the Microprocessor Systems students and want to point out that even though I was teaching the course material to them as the teaching assistant, I learned as much from them as they learned from me. I am grateful to the Fonds Québécois de la Recherche sur la Nature et les Technologies for their financial assistance during my degree.

Finally, I wish to thank Andréanne Moreau for her constant support and encouragement during my master's studies.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.1.1 Overview

The digital age is a period in human history characterized by an economy based on information computerization. It is allowing rapid global communications and networking to shape modern society. It is also important to note that the information age has been formed by monumental advances in microelectronics technology. As the fabrication process and technology used in the semiconductor industry improves and engineers find ways to fit more and more transistors together on a microelectronics circuit, computing power grows exponentially year-to-year. Today, modern microprocessors offer high processing capabilities at very low cost. For example, recent ARM-based microprocessors can offer similar computing performance to what was present in personal computers 15 years ago but at a significantly smaller form factor and energy consumption. As the cost of computing decreases, it is now economically feasible to integrate more computing capabilities in new fields of activity.

In many end applications such as robotics and industrial control, computer systems need to provide a very high degree of precision as well as quick and deterministic reaction times. At the same time, there is also a desire for these systems to provide a higher degree of intelligence and Human-Machine Interaction (HMI) features. General purpose computer systems can usually provide the former but cannot provide the latter. Building a solution that can address all of these needs while keeping costs and complexity low can be very challenging.

This thesis proposes using modern microelectronics technology to address the challenge of unifying general computing features such as video output, networking, and HMI with the ability to provide quick and deterministic reaction times in an easy-to-use computer architecture. The solution takes the form of a computer system on which it is easy for an engineer to build software that uses both types of capabilities.

This development platform is developed in tight collaboration with *MotSAI Research* as well as with funding from the government of Quebec and the government of Canada. The project's name is *Golden Lion* and will be used as the basis to a set of controllers that can be used on industrial machines, high-precision control systems, and other similar fields.

1.1.2 System-on-Chips

Customers who purchase a computer of any form, whether it is an on-board computer in a car or a home PC, demand as many functionalities as possible from the products they buy. This has pushed the microelectronics industry to integrate as many features as possible into a given electronic circuit, as opposed to building circuits that are dedicated to a small number of roles. This trend has produced what is called a System-on-Chip (SoC), which is an electronic chip that can perform a large number of roles concurrently, whether it is communicating over the internet, producing a video output to a screen, or performing the role of a computer's CPU, the central "brain" of a computer. The advent of SoCs allows computer engineers to build computer systems that can perform more functions and use less electronic parts to do so. This technology is at the core of the *Golden Lion* project and allows it to contain many features while retaining its small form factor and reduced complexity.

1.1.3 Development Environment

A development environment is a set of software tools that help a computer programmer create programs that run on a particular platform. These environments reduce the time it takes for a programmer to create fully-functional programs. As SoC manufacturers integrate more features into smaller form factors, the inherent complexity of these devices increases. This makes it more difficult for developer who uses these devices in their products to get them to the market in reasonable time frame. To address this difficulty, manufacturers provide software tools that abstract away the technical details of developing for their

platform. It has become the standard for manufacturers to provide intellectual property and design examples to allow a developer to build a system in weeks as opposed to years.

Many vendors leverage established standards so that software developers can use familiar development environments and get it up-and-running quickly. For example, in the field of embedded systems engineering, GNU/Linux is set to become the operating system of choice on embedded SoCs due to its vast array of established software libraries and tools.

Furthermore, more and more engineers want to design and test their real-time algorithms using a technique called HIL simulation. HIL simulation is a technique where the development and testing of algorithms is done using an actual actual plant as opposed to a simulation of it. It can also involve the use of a embedded controller interacting with a plant simulation. By testing a design with actual hardware, using this technique can reduce development time and time to market.

Golden Lion stands on the shoulder of giants. By using Free and Open-Source Software (FOSS), it has been possible to create a functional prototype for *Golden Lion* within a small fraction of the time it would take had all the software been written from scratch. FOSS is free to use, distribute, and modify. It has lower costs, and in most cases costs only a fraction of their proprietary counterparts. FOSS also allows programmers that use *Golden Lion* for their projects to be familiar with the tools right away and help them finish their project as quickly as possible.

1.2 Actors

It is important to define the main actors that are referenced in this thesis. The objective of the project is to design the flexible baseline of a computer system that developers and integrators use to build a working solution for an end-user. Developers, engineers and integrators use the base design to build their applications. They are also referred to as the *user* of the baseline system and its suite of development tools. The operator is the person that uses the end-application of the computer system that the integrator has built.

1.3 Contributions

In this thesis, the basis for an industrial programmable controller is designed using the latest SoC FPGA technology. A hardware and software architecture has been devised to

fulfill this purpose and is compared to other existing computer architectures. The main academic contribution is the demonstration of open-source simulation software performing HIL simulation on a SoC FPGA. An analysis of the performance of the HIL testbench in comparison to other setups is performed. Accompanying this is a novel command-line debugging tool to assist programmers in developing their application.

1.4 Thesis overview

Following the introduction, this thesis will give an overview of existing real-time and co-processor architectures as well as techniques used to optimize performance of these systems. Various aspects such as the design flow related to real-time controller development will be also examined.

Chapter 3 will discuss the architecture design decisions and trade-offs. The quantitative and qualitative requirements of the platform will be elaborated upon. Design decisions involving both the software and hardware portions of the system as well as issues related to the particular use of the operating system will be also be covered.

Chapter 4 will cover the details of the design of some of the system's IP blocks. The protocols used in the experiments as well as the basic setup will be elaborated.

In Chapter 5, the various development environments to help engineers create their applications on the *Golden Lion* platform and their associated workflows will be explored. This chapter contains the main contributions of this thesis. It will be shown how open-source tools can be used in the context of HIL simulation.

Finally, the last chapter will contain the conclusion and suggestions for future work as well as potential improvements to the real-time architecture and development environments.

Chapter 2

Background

In this section, the background work on systems related to various components of the project's architecture will be reviewed. First, there is an overview of the published works on how to deal with evolvability in industrial system design, which is one of the main intended end applications for this platform. Next, an overview of the technology of computer buses will be performed. Today's modern SoC FPGA are then reviewed and analysed in terms of what features they offer to the development of this platform. One of the basic IP cores designed for this thesis is an LMS-based adaptive filter, which is why the basics of adaptive filtering are covered. A brief overview of techniques in hardware/software co-design are explained. Following this, an overview in the design of co-processors as well as their associated connectivity to the main processor is shown. Finally, the documentation and publications related to the various tools used in embedded system development are presented. This includes a review of Model-based design (MBD) software as well as techniques such as HIL simulation.

2.1 Dealing with evolvability in embedded system design

The methodologies for embedded system design have changed continuously as technologies available to the designers have improved. With that improvement of technology comes additional complexity. This can sometimes cause an increase in development time.

The lifetime of an embedded system often does not end at delivery. In many applications, such as in industrial electronics system design, there are many situations where there is a need to add new features or apply new requirements after the first deployment. The

reality is that embedded systems exist within the constraint of economics. It would not make sense for an owner of a system or a system maintainer to completely overhaul their machines every time there is an evolution in technology. Sometimes, using new technology results in under-the-hood improvements that are not always immediately apparent to the user or do not always yield immediate monetary results. Legacy industrial systems are becoming more and more common as it is costly to constantly upgrade industrial systems. Obsolescence is also a big problem as it can get very costly to replace parts that are no longer fabricated, obliging maintainers to keep a stock of spare parts. A decision process has been proposed to deal with the increasing challenge for owners of stable working systems to deal with the obsolescence and phasing out of older parts [1]. Even algorithms [2] and simulations [3] have been developed help estimate the costs of part obsolescence and plan for the related eventualities. Researchers are also looking at FPGAs to approach this problem [4]. This approach makes sense, because Intellectual Property (IP) blocks like an embedded CPU can be programmed in an FPGA and this IP block can be more easily transferred to a more modern FPGA when this part becomes obsolete. This way, the same functionalities can be achieved and there is very little porting work to do.

Papers have also been written on how to manage the evolvability of embedded systems where the management of legacy technology becomes inevitable. For example, there are papers that discuss how to design an embedded system with evolvability in mind by using an approach based on concept creation and buffer interface wrappers[5].

Hallmans et al. discuss this problem [6] and propose a methodology for handling evolvability in complex embedded control systems that are expected to function for a long time [7]. The authors present a method to minimize the work required to maintain existing functions as well as implement new ones. They specifically identify two major challenges when dealing with maintaining embedded systems over a long life cycle. The first problem is of obsolete components, which makes it difficult to repair or even prepare for new *features* to existing systems. Here are the following options for dealing with this: You can choose to buy components for the rest of the system's life cycle, you can replace the broken component with a new one, or if that is not possible, you can replace a larger part of the system. The next challenge is the capability to accommodate new *requirements* on top of the existing installation. The authors specifically present the issue of cybersecurity as an example: when increasing connections to the internet are established in the workplace, it becomes necessary to protect these systems against possible vulnerabilities arising from

these connections. In addition to these two difficulties is the issue of having a structured for *testing and verification* once new features are applied.

Here is an interesting component of their methodology: One enumerates all of the requirements of a unit along with its dependencies and its included functions. This includes the functions (filters, amplifiers, Analog-to-Digital Converter (ADC)s, User Interface (UI), etc) and their associated resources: configurations, communication (data buses, external buses, communication interfaces), Input/Output (I/O), and hardware platforms. With this enumeration, one associates each requirement with certain function as well as its set of supporting resources and build a dependency graph such as the one in Figure 2.1. By representing these functions in a graph, you can more readily determine the impact of the failure of a component and the implications of an eventual upgrade or repair.

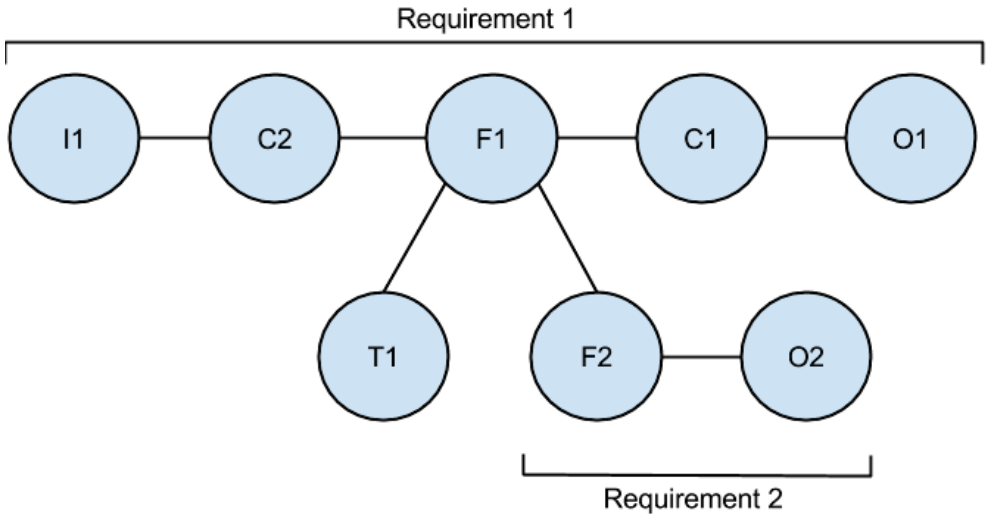

Fig. 2.1 Example dependency graph of an embedded system with its associated requirements

F: Functions

T: Configurations

C: Communications

I or O: Inputs or Outputs

H: Hardware

They also describe how this methodology can be used with functions and its associated resources in release handling. This is useful since it is important to keep track over time which functions are included in different releases and versions of a product. Their proposed release handling method deals with this by immediately identifying possible conflicts in a dependency graph. It also allows a company to identify which versions of their devices need to be verified for a particular function F.

2.2 Computer Buses

In computer architecture, a computer bus is a communication system used to transfer information between components of a computer. It can also be used to transfer information from one computer to another. Generally, a computer consists of a CPU to perform mathematical operations on data, main memory to keep hold of that data, as well as a variety of peripherals to send that data back and forth with the world outside of the computer.

Modern systems can have more complicated architectures and can include a larger number of components. For example. multi-core CPUs, primary and secondary hard drives, Graphics Processing Unit (GPU)s, networking peripherals, and screen displays all need to communicate with each other. Different types of buses are used depending on the requirements of the particular application. Generally, buses connecting closely placed components are more performant than ones connecting to components that are far apart. For example, on-chip buses that connect a CPU to its closely placed peripherals such as the cache and fast memory components that are on the same die are several orders of magnitude faster than buses that connect a CPU to peripheral cards. Over the years, there has been a lot of effort spent by researchers in improving the bandwidth and latency of different buses. As a result, bus design has taken on many different forms and has become more complex. Depending on the different applications, buses can be serial or parallel, synchronous or asynchronous.

In the 1990s, most of the computer peripherals were connected to the CPU through a bus standard called Peripheral Component Interconnect (PCI) [8]. PCI started out as a 32-bit bus that operated in the range of 33 MHz that could be upgraded to accommodate 64 bits. As technology improved over the years, the operating frequency has also increased. The bus could only handle a maximum of five devices at a time. However, using a parallel bus has its limits.

As it turns out, serial transmission is faster than parallel transmission due to the number of bottlenecks involved [9]. One of the problems with parallel transmission is the phenomenon of crosstalk, where communicating parallel wires interfere with each other, either through attenuation, induction or cross-coupling. With these errors, it becomes necessary to process them at the receiving end, which increases the overall overhead. Another problem is that it cannot be guaranteed that all of the signals going from the transmitter to the receiver will arrive at the same time. This means that the operating frequency is only as fast as the slowest of the signal lines. This synchronization problem is made worse when the signal lines go through long distances (such as in desktop motherboards) and the length differential between the signals lines is exacerbated.

Because of this, the computer electronics industry has shifted from using parallel transmission to serial transmission technology such as Serial ATA (SATA) or PCI-e. Increasingly, PCI-e is becoming the standard for high-performance communication between a computer's CPU and its peripherals. It is the successor to the the parallel PCI and AGP serial bus standards. The improvements of PCI-e over PCI are striking. Because of the parallel nature of PCI, the speed of the bus was limited to its slowest connected peripheral and was limited to one master at a time in a single direction. In contrast, PCI-e supports full-duplex two-wire serial communication between any two endpoints with no limits on concurrent accesses between multiple endpoints on the PCI-e interconnect. Each two-wire serial connection is called a PCI-e *lane*. Each lane can operate as fast as 4 Gbit/s [10]. To increase the bandwidth on a PCI-e connection, the number of PCI-e *lanes* to the peripheral can simple be increased. The superiority of PCI-e can be explained by the fact that at high frequencies, it is easier to make a single connection go 16 times faster than to double the speed of 8 connections. On the software side, the PCI-e standard uses a packet based protocol for the transaction. From the point of view of the CPU, the Operating System (OS)'s kernel detects PCI-e endpoints with their associated Base Address Register (BAR) which allows for communication through memory access instructions.

For the last decade, designers have been trending towards Network-on-Chips (NoCs) topology for internal communication between nodes in an Integrated Circuit (IC). Even in 2002, researchers were seeing NoCs as the next thing in SoC design [11]. NoC technology is a communication system that is present inside modern ICs and is used for interconnecting different IP cores. It can be used to connect modules that may be in different clock domains. Bridging different clock domains is important, since modern ICs no longer can rely on only

one clock signal to remain synchronized and are vulnerable to clock skew effects. For on-chip communication bus structures, the present trend concretising is towards a NoC-based topology. There are publications that propose NoC topology in a design. For example, Dally and Towles propose replacing design-specific global on-chip wiring with a general-purpose on-chip interconnection network. They say that their approach has advantages in structure, performance and modularity. It results in significantly lower power dissipation, higher propagation velocity and higher bandwidth than what is possible with conventional circuitry [12].

2.3 FGPA-based System-on-Chips

2.3.1 FPGA

An FPGA is an integrated circuit that can be configured by a designer after manufacturing. It implements digital logic that is specified by a designer using Hardware Description Language (HDL). An FPGA can contain programmable logic blocks as well as a hierarchy of reconfigurable interconnects that allow the blocks to be connected together. The big advantage of using FPGA in designs is flexibility and post-release updates to add new features or to fix bugs. Using an FPGA for a digital logic design as opposed to an Application Specific Integrated Circuit (ASIC) is that it shortens the time it takes to get a product to market. There are many companies in the business of manufacturing FPGAs. The main competing vendors in this space are Xilinx, Altera and Microsemi (formerly known as Actel). Devices specialties range from low-cost chips intended for embedded applications, to very expensive high-end chip that can deliver a lot of processing power intended for telecommunication or intensive video processing applications for example.

2.3.2 SoC-FPGA

Electronic integration of multiple functions into single chips has been undergoing ever since the creation of the IC. Today, this trend persists within the domain of FPGAs with the introduction of the SoC FPGA. A SoC FPGA combines the flexibility of programmable logic with the performance of an ASIC. The most popular examples of this type of circuit are the Xilinx Zynq family and the Altera SoCFPGA. Both these product families contain ASIC processor blocks based on the ARM Cortex-A series of processors alongside their

collection of peripherals. The Cortex-A series is an embedded processor IP block sold by ARM Holdings [13]. It is used in a wide variety of applications such as mobile and embedded end uses. It posesses the standard components most CPUs should have, such as multiple levels of cache memory, multiple cores, a 32-bit Instruction Set Architecture (ISA). Its vendor, ARM Holdings, is a fabless company whos business model is based on selling their CPU blocks to third-parties who take care of the actual manufacturing. By integrating these CPU blocks into the same die as the FPGA, FPGA designers help system designers use less electronic components in their solution. By using less electronic parts in a design, it can allow for lower power consumption and cause less integration headaches.

2.3.3 FPGA Tools

Although previously very expensive and costly to develop, PCI-e IP cores have become standard in the development suites distributed by the main FPGA vendors [14] [15]. Being imprinted onto the silicon dies and connected to Serializer/Deserializer (SerDes) high-speed serial transceivers, these 'hard' IP cores make it much easier to develop applications that are accessible by PCI-e. Accompanying this IP is a set of HDL files that provides the verification engineer with an application programming interface (API) to help simulate the behaviour of a PCI-e endpoint. Research regularly leverages this technology in experiments. For example, Cao et al. use the Xilinx Development tools and free-of-charge IP cores to run a PCI-e root complex on one development board as well as an endpoint on another one. It includes an AXI-PCIe bridge which translates the BAR to AXI protocol. They have also written device drivers to control the root complex and ultimately they want to control other SOPC devices with this method [16].

An important component of the suite of development tools from FPGA vendors is the system-level design tool. This sort of tool allows designers to create their FPGA design at the top level by binding together IP cores that are either created by the designer or by the FPGA vendor. In the case of Altera's tool, QSys, it gives designers access to an automatic NoC generator and its associated bus translators that bind together the IP cores in the system.

2.4 Hardware/Software Co-Design

Hardware/Software Co-Design (HSCD) is any design technique for the creation of systems with both hardware and software components. At the core, it is a technique that uses the synergism of hardware and software design to meet system-level objectives. Embedded systems design is particular because the hardware and the software must be designed together to make sure that the solution meets performance, cost, function and reliability goals. It requires delicately balancing of software and hardware resource requirements. Cost is an important consideration when choosing either hardware or software for the implementation of a product [17]. The technique has evolved over the years and has readily taken many forms as researchers have been looking at ways to improve the workflow associated with this technique.

Even in the 1990s, the potential of FPGAs for HSCD was being recognized. In an invited paper on HSCD published in the IEEE proceedings in 1997, De Micheli and Gupta explain how FPGAs are game changers in the context of HSCD [18]. FPGA technology can be reconfigured after manufacturing to perform tasks better than if they had been performed on a microprocessor. This can be done without changing the underlying hardware as opposed to digital design using Very Large Scale Integration (VLSI). In this paper, they say that the main challenge of HSCD is identifying the crucial portions of software algorithms and implementing them in either hardware or programmable hardware components [18]. The first part of the challenge is addressed by determining if it is worth porting a particular section under bandwidth and communication constraints. Although primarily done by humans at first, there are papers and software tools that present algorithms that produce hardware/software partitions [19] [20]. The second part is addressed by generally available synthesis tools that are usually available from an FPGA's vendor.

2.5 Memory-Mapped Co-Processors

A computer system designer can optimize the execution of different types of algorithms by making hardware modules available. When algorithms are executed in specialized hardware modules as opposed to being executed on a general purpose processor, the execution speed is often increased by orders of magnitude. There are numerous papers that explore the use of co-processors to speed up the computation of specific algorithms. An important issue

is the mechanism with which a computer system's CPU core communicates with hardware co-processors and vice-versa.

2.5.1 Co-processors

Research in co-processor design is not recent. There are many publications from the 1980s and 1990s that discuss off-loading mathematical operations to a separate digital circuit. In 1990, Chu and Yaohan published a paper that discusses the direction and areas of interest in the design of co-processors. They define coprocessor computers as application-specific, subordinate processors that work jointly under the control of the main processor[21]. They give the example of using co-processors to hasten the computation of floating-point operations. This is something that Diodato et al. experimented with when they published an academic paper on the design of a single-chip VLSI co-processor that performs floating-point operations[22].

Co-processors exist for a wide variety of end applications. Their roles can be as specific as providing a parsing engine for compilation [23] to display graphics accelerators. Display and graphics computation consists of a large portion of the motivation for co-processor design. The need for graphics display has been present since the appearance of screens for personal computers. As early as 1987, semiconductor designers have been designing graphics accelerators for consumer use [24].

There are also many other publications that mention the use of co-processors to offload specific algorithms to improve processing time where *real-time* performance is important. In the late 1980s and 1990s, the use of co-processors in applications that require real-time performance such as communications [25], medical imaging [26], real-time task scheduling [27][28][29], real-time image processing[30] and compression [31] [32], kalman-filtering [33] and signal processing [34][35]. Real-time applications also include video decoding for consumer PCs. Video decoding is also a useful target application for co-processors due to the heavy weight of this type of computation and the improvement to the overall performance it can bring. As a result, a lot of research has gone towards designing MPEG-4 decoders. MPEG-4 was a common coding standard for multimedia applications. Berekovic et al present the design of an MPEG-4 video VLSI decoder [36]. Today, co-processor products are appearing for a wide range of end-applications, from small embedded devices to high-end server applications [37]. A big development in the field of server-side computing is the

incorporation of FPGAs as a co-processor to optimize certain portion of search algorithms in Microsoft's Bing search engine [38].

2.5.2 Memory-Mapped Interface

Memory-mapped I/O is a method for performing bi-directional information transfer from a computer's CPU to its peripherals. Using this technique, I/O devices are accessed using memory instructions that access the same bus as the rest of the system memory. As a result, overall system design is simplified and can result in faster and simpler hardware. Comparatively, port-mapped I/O uses a special category of CPU instructions which are dedicated to accessing I/O devices and have a separate address space. Port-mapped I/O can sometimes be called *isolated I/O* because it is isolated from main memory. This is the case for some of Intel's processor chips [39] for example. Most co-processors need to interface with the main CPU. The concept is not new. For example, the floating-point accelerator IC designed by Diodato et al[22] is accessible from the CPU by memory-map.

A good deal of research has been done to examine the effects of the choice of interface on the performance of the system. For example, Hodjat et al. examine the situation of interfacing a LEON core with a custom cryptographic processor. [40]. The authors explore the interface choice to the encryption co-processor to optimize the cycle count, throughput, LUT usage and energy usage. They consider both the Custom Peripheral Interface (CPI) and a memory-mapped interface (through an AMBA bus). The authors conclude that using the CPI to interact with the AES core is much faster than having a memory-mapped interface.

An important use case for memory-mapped co-processors is dealing with a large amount of information transfer. One way for the CPU to transfer information through a peripheral's I/O ports is to use dual-port BRAM to directly map the inputs and outputs of peripherals to the computer's memory space using a Wishbone interface [41]. This provides the advantage of not having to implement a direct memory access (DMA) system that gives the CPU access to the information. As soon as the incoming bytes arrive, they are immediately made available in the memory-map.

Memory-mapping is not restricted to embedded systems. Due to the large amount of file handling in server database systems, a lot of research has been performed to examine the different ways to manipulating hard-disk data. Song et al. examine the performance

differences between using mmap() to perform I/O on a file handler as opposed to using the traditional read() and write() when performing data-intensive applications [42]. The authors suggest improvements that would help reduce latency and throughput by modifying the Linux kernel. As it turns out, there are differences in performances for both approaches depending on the actual size of the file. For large files that exceed the size of the physical memory, the I/O performance advantages of mmap() are not present. The reason for this is because the virtual memory system does not take advantage of the features of modern flash storage devices. In essence, using Memory-mapped Input/Output (MMIO) for a large data sets does not scale well. The authors suggest an improvement to the situation by suggesting multiple improvements to the Linux kernel. They present a solution that guarantees a performance that matches traditional file I/O using read() and write() in cases where there are cache misses for the memory calls and they also guarantee improved performance for situations in which there are cache hits. The main optimizations come from reducing the overhead of the virtual memory subsystem for file-/IO by doing multiple inter-processor interrupts instead of doing per-page IPIs. The results show the most improved throughput for random reads and slightly improved throughput for sequential reads. The experiment also shows reduced latency for random reads and sequential writes.

2.6 LMS-based Adaptive Filters

There is a major advantage to implementing an algorithm such as a filter on an FPGA as opposed to a CPU, due to the easily exploitable parallelism of the filter. It can take a CPU a large amount of clock cycles to perform the same amount of signal filtering that an FPGA can perform in just a few clock cycles. An important type of filter used in many applications such as communications and signal processing is that of the adaptive filter.

2.6.1 Adaptive Filters

In many applications, it may be necessary to adjust the response of a given filter due to continuously changing conditions. Adaptive filters address this. The objective of adaptive filters is to modify a given FIR filter so that its output signal $d^*(n)$ matches a desired signal $d(n)$. To modify the FIR filter, the coefficients are updated with the use of an adaptive algorithm. The generic adaptive model is shown in equation 2.4, where each filter tap w converges to a desired value over time. In essence, the input signal is distorted by a given

physical channel used for communication as well as other factors such as the quantization of the input. The system is modeled as an FIR filter h_c^H which takes in the input samples $x[n]$.

The variable filter itself is a p order FIR with coefficients:

$$w_n = [w_n(0), w_n(1), ..., w_n(p-1)]^T \tag{2.1}$$

The error signal is

$$e(n) = d(n) - d^*(n) \tag{2.2}$$

where $x(n)$ is the input samples vector:

$$x(n) = [x(n),\ x(n-1),\ ..., x(n-p+1)]^T \tag{2.3}$$

The coefficients of the FIR filter are updated with the means of a correction factor $\triangle w$, which is determined by the chosen adaptive algorithm:

$$w_{n+1} = w_n + \triangle w \tag{2.4}$$

The system requirements are what dictate the choice of the specific adaptive algorithm to use (LMS, RLS, BLMS, etc) as well as the length of the FIR filter. A large number of coefficients are used when the channel has a high distortion rate. However, it is the rate of change of distortion that affects how fast the filter has to readjust itself and if parallel or serial updating of the filter coefficients is needed. So technically, you can end up with a filter with a large number of coefficients but since the variation in the system is slow relative to the system clock, a serial updating of the taps can be used to reduce area and power usage.

The main optimization points in adaptive filters are the rate of convergence, the resource utilization, and the power consumption. You can often trade off area usage for speed of convergence. A parallel versus serial implementation is exactly this, where the speed of convergence of a parallel implementation trumps the serial one, but the resource utilization is much higher. The only way to make a serial implementation as quick as a parallel implementation is to increase clock speed, which increases power consumption.

2.6.2 Least Mean Squares Algorithm

The basic idea behind the Least Mean Squares (LMS) filter is to use the gradient descent to converge to the optimal filter weights. The algorithm starts off by assuming the filter weights are small, and determines the change of filter taps using the gradient of the mean square error.

$$w_{n+1} = w_n + \mu x(n)e(n) \tag{2.5}$$

The main advantage of LMS is its low complexity of computation relative to other adaptive algorithms such as the Recursive Least Squares Filter. However, it is not the algorithm that converges the most rapidly, which is needed in some high-speed applications. For the interested reader, a more thorough analysis of this type of filter was performed in the 1970's by Widrow and Hoff [43].

2.7 Development Environments

In the general sense, a development environment is a set of programming tools that allows a developer to build applications for an intended target application.

2.7.1 CMSIS-SVD

The CMSIS System View Description (SVD) format formalizes the description of the system contained in many of ARM's microcontrollers. The information ranges from high level functional descriptions of a peripheral all the way down to the definition and purpose of an individual bit field in a memory mapped register [44]. It is arranged in an Extensible Markup Language (XML) format. The hierarchy is represented in Figure 2.2. The top-level element is the device that is being programmed on. The next level is the set of peripherals (UART, I2C, HDMI, DMA, etc.). Inside each of those peripherals are registers used to observe and control the behaviour of the peripheral. Each register has a set of fields which represent a particular control value, which is located at a predetermined bit offset in the register. The fields can take a range of possible values that are associated with a description of the behaviour called enumerated values (Ex. Bit EN = 1 (On), EN = 0 (Off)). Vendor specific formats can also be added to the hierarchy if needed.

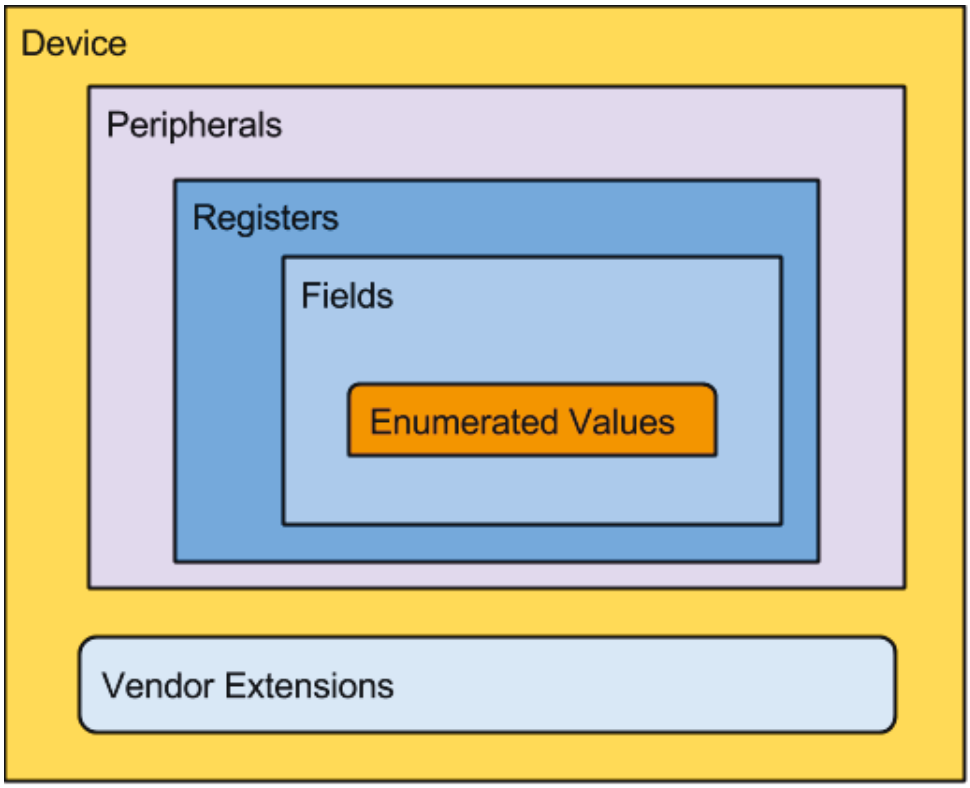

Fig. 2.2 SVD file format hierarchy

The SVD format is mainly used as a method to standardize the way information about a register map is represented to an ARM CPU. It is mostly used by silicon to help development tools provide a comprehensive debugging environment in Integrated Development Environments (IDEs). This format is also used in the context of describing the contents of the memory-mapped devices programmed inside an FPGA. On Altera's platform, when the QSys tools generates HDL describing the system in the FPGA, it also creates an SVD file. IP component designers have to create their own SVD file to become a part of the final documentation about the internal register map.

2.7.2 Python

Python is a general-purpose scripted programming language that is friendly and easy to learn [45]. The language is ideal for quickly programming prototype applications, but it also provides constructs that can be used for building large-scale applications. For example,

in an application note written by Jean-Samuel Chenard on using Userspace I/O (UIO), Python is used for quickly prototyping memory-mapped peripherals [46]. It integrates very easily with modern POSIX-based OSs such as Linux and OS X. Because of its portability, it is also often used in embedded Linux systems of all kinds. It possesses a large standard library, which is often referred to as its greatest strength, as well as an official repository of user-build packages called *Pypi*. Python contains modules that can connect to Graphical User Interfaces (GUIs), regular expressions, unit testing, etc. It also possesses widely used OS system calls. The programmer can even program critical software components in C and use it by linking to it from Python.

2.7.3 Scilab

Scilab is an open-source software for numerical computation. It is similar to commercial simulation software such as MATLAB [47]. It includes hundreds of mathematical functions and can be applied to many fields such as control, simulation, optimization and signal processing. [48] introduces Scicos for the first time as an extension of Scilab. It is presented as a system builder that incorporates both discrete and continuous-time components. It introduces the basic blocks such as the fundamental discrete-time event block where the outputs equal a function of the inputs that is updated at an event signal, the static block where the output is simply a function of the inputs, the event generator blocks, etc. They regularly use Simulink as reference guide in, for example, use cases and user interface.

2.7.4 Graphical algorithm design with Xcos

Xcos is an extension of scilab as a graphical user interface for systems modelling and simulation [49]. Models can be designed, loaded, saved, compiled and simulated. Xcos is freely available with Scilab. It is a tool dedicated to the modelling and simulation of hybrid dynamic systems which include both continuous and discrete simulation models. The editor allows the representation of models as block diagrams that can be connected to each other. Blocks can represent models designed by the user or that were pre-shipped with Scilab.

There are papers that have been written about the use of Xcos for professional development. For example, Guezar et al. explore the use of hybrid simulation in Scicos [50]. In fact this is the paper that introduced simulations in Scicos, which is the ancestor of Scilab

in this respect. In particular, they want to simulate hybrid systems, which are systems that have both discrete states/events such as state transitions and continuous elements, such as a temperature value in a room or a charge on a capacitor. In particular, they show that they are able to simulate a voltage booster.

2.7.5 Hardware-in-the-loop Simulation

HIL simulation is when simulation software is executed with a mix of simulated an actual hardware components. There are control interfaces that allow communication between the simulation software running on a desktop computer and external hardware. This simulation is used to test an electronic component such as a microcontroller or an FPGA (in the field is usually called an Electronic Control Unit (ECU)), which instead of being connected to the real equipment under control, it is actually connected to a simulation. The general architecture of a HIL testbench setup is shown in Figure 2.3.

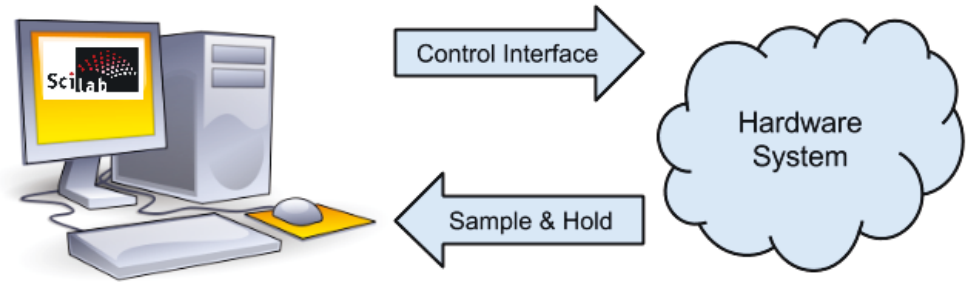

Fig. 2.3 General Architecture of an HIL testbench

HIL testbenches are generally used in the context of the development of real-time embedded systems. Instead of testing the algorithm of an embedded system on an actual plant, the system is tested on a plant simulated on a computer. This can provide significant cost savings for developing and testing embedded applications. HIL simulation is a technique that was first used mainly in the aerospace industry as a means to verify and validate the algorithms of cyber-physical systems [51]. It is used in an increasing number of fields of engineering such as the automobile industry to test active breaking systems, traction control systems, electronic stability programs [52] [53] as well as autonomous driving programs [54]. This technique does raise some challenges such as complex interfacing requirements

and the ability of the simulation software to accurately replicate a real plant and execute within strict real-time deadlines. However, even with relatively weaker processing power of computer in the 1990s, this technique has shown that it can lead to reduced development time and time to market. It has also been shown that it can help ensure the safety and reliability of electronic components that run complex algorithms [51].

There are papers that have been published that use HIL testbenches to test the performance of scheduling algorithms [55] when the communication is under the effect of jitter. However, there are not many publications that examine the different communication mediums between the computer and the control interfaces. This thesis will try to address this by proposing a HIL simulation testbench that is suitable for use with SoC FPGAs and analyze its performance and reliability for algorithm development.

Chapter 3

System Architecture

In this chapter, the overall requirements of the Golden Lion computer system as well as the design decisions leading to its final form are explained and compared with related works. An analysis of its flexibility and its evolvability is performed.

3.1 System Requirements

The main system requirement for this project is to have a computer system that can perform low-level capabilities such as fine-timing manipulations and high-throughput processing that only an FPGA or an ASIC can provide. At the same time, it is also a requirement for this computer system to offer high-level processing capabilities such as HMI and networking. The high-level and low-level functionalities need to be tightly integrated without the possibility of them interfering with each other. The failure of one portion of the system should not cause the failure of the other. For example, a crash in the system's UI program should not have any detrimental effect on the strict real-time functionalities of the system. There is also a need for the system to be inherently modular and it has to be simple and easy to add or remove IP blocks inside the FPGA co-processor.

3.2 System Overview

The Golden Lion project is designed as a two-level processing architecture which effectively segregates the real-time demands of an application from the soft-time demands such as networking, monitoring and HMI. The platform will rely on proven computer archi-

tecture fundamentals, such as structured memory access, cache coherence and memory protection mechanisms. These foundations will contribute to strengthening the system's robustness. One of the core fundamentals of this platform is the separation of computing resources between its real-time portions which deal with tight response constraints and non-real-time capabilities which deal with a higher level of control. This allows for the segmentation of a problem between its more relaxed soft real-time requirements and hard real-time constraints. The hard real-time portions of the system are handled by a memory-mapped FPGA co-processor. The key to the success of the implementation is the unified memory mapping between the main processor and its co-processor(s).

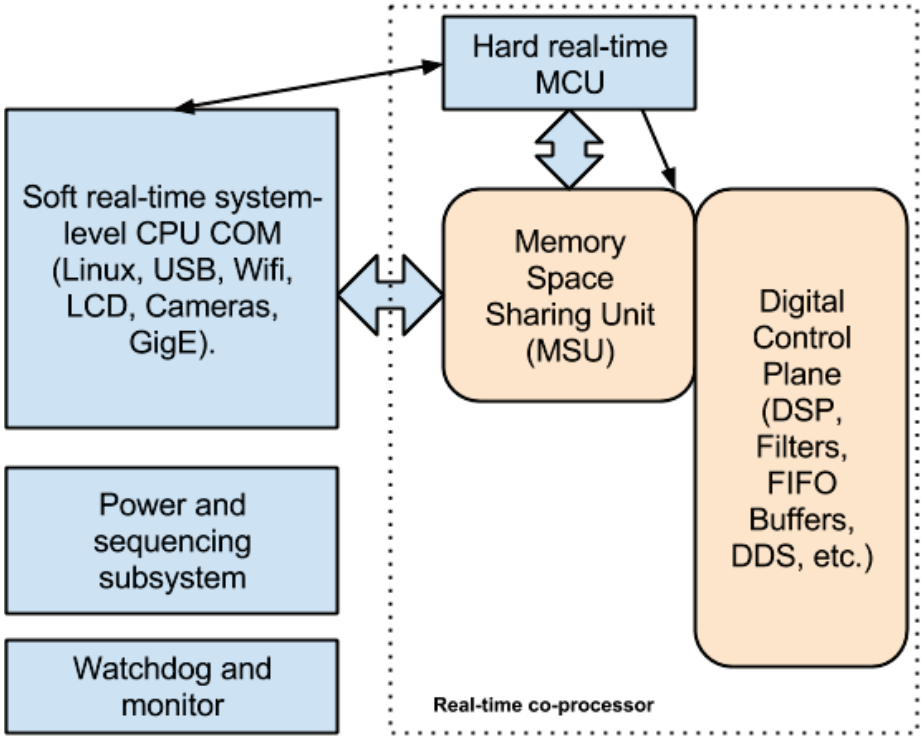

Fig. 3.1 Golden Lion Overall Structure

The system consists of a high-performance Linux system which is expected to have enough RAM, flash to offer decent UI application as well as solid networking and graphics. The hard real-time microcontroller will have the role of executing tasks with hard real-time requirements alongside the FPGA. The FPGA will be used to control the I/O devices, First In First Out (FIFO) buffers, and MotSAI's IP cores that control the backplane pins. Communication between the microcontrollers and the FPGA will be performed with the

use of a memory-mapped interface connected to a shared memory space. Other vital safety mechanisms such as watchdog timers and monitors as well as a power subsystem will be added.

3.3 Initial Design Decisions

3.3.1 Prior platform

Prior to the start of the *Golden Lion* project, MotSAI Research had developed a platform consisting of an LPC microcontroller which communicated to an FPGA through an serial peripheral interface (SPI) port. The role of this platform was to generate analog and digital output signals. Although the final product addressed the client's needs, MotSAI would need to improve the design in order to create an all-encompassing solution for a broader range of applications.

3.3.2 Initial Proposal

With the experience gained from the design of the previous board, MotSAI proposed the design of a new system which still consists of microcontrollers and an FPGA, but this time is connected by a memory-mapped interface. At the start of the project, the initial proposed architecture consisted of an STM32F4 microcontroller and an Freescale i.MX6Q processor both connected to an FPGA. This formed a decoupled architecture with multiple processing points. The main difference between the prior platform and this one is that the microcontrollers would interface with the FPGA through a memory-map interface instead of through SPI. Using a parallel memory mapped interface as opposed to a serial bus like SPI to communicate with the rest of the system has the advantage of greatly increasing the possible bandwidth and reducing the latency.

The Freescale i.MX6Q microprocessor was chosen as the high-level microprocessor in this project. It has an ARM Cortex-A9 quad core processor that has a high clock rate (1.2 Ghz) and ample cache memory (32 KByte instruction and data caches as well as 1 MByte L2 cache). It supports a wide range of diverse features such as a graphics acceleration, image processing as well as display and camera interfaces. The image processing is of interest due to the potential image processing applications that MotSAI intends to target. The support for external busses is also very diverse. Not only are basic serial busses like inter-integrated

circuit (I^2C), SPI and universal asynchronous receiver/transmitter (UART) featured, but more advanced busses like PCI-e 2.0 and universal serial bus (USB) are also supported. Monitors, real-time clocks (RTCs) and security features are also standard on this chipset.

In the system, the STM32F4 has the role of the real-time microcontroller. Its available features include an ARM Cortex-M4 processor with a high clock rate (168 MHz) and a floating point unit (FPU). It also includes peripherals such as USB, Ethernet MAC and serial peripheral busses like I^2C and SPI. The microcontroller also includes a flexible static memory controller (FSMC) peripheral that allows accesses to external peripheral memory. This controller is what was going to be used to talk to the FPGA.

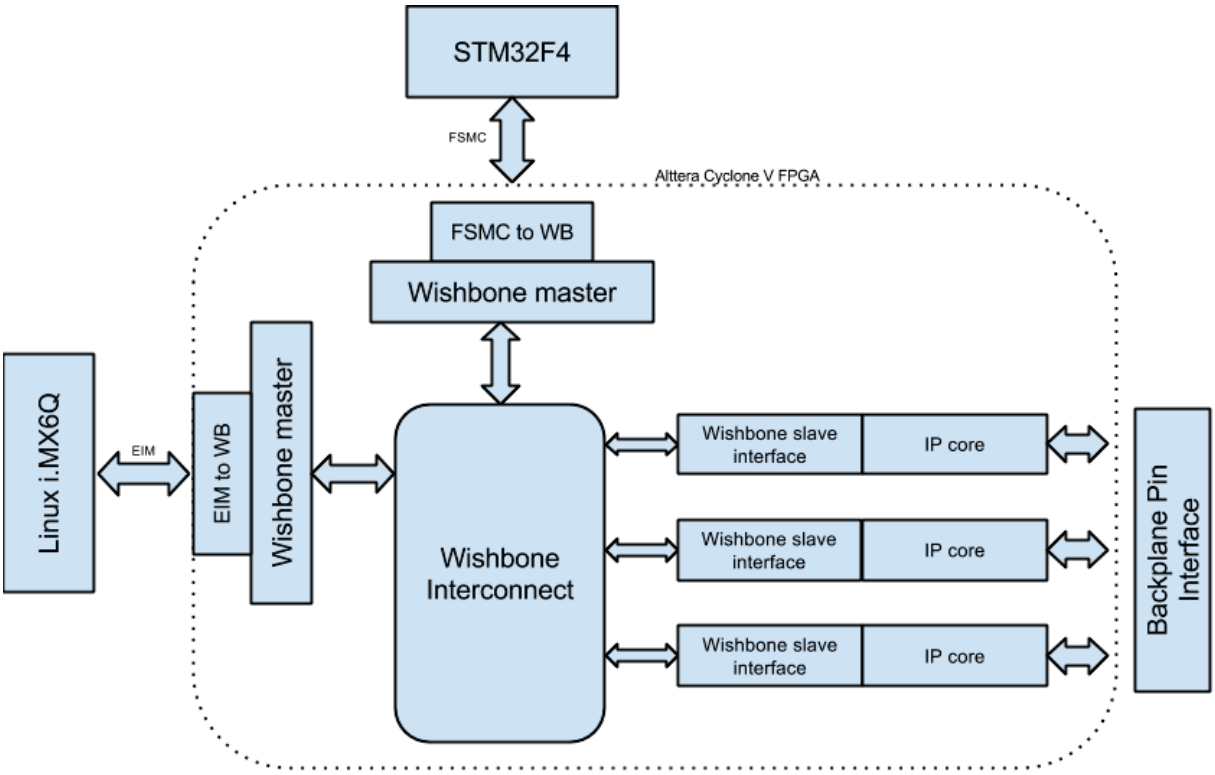

Fig. 3.2 FPGA Internal Structure

The internal structure of the FPGA is represented in Figure 3.2. The inside of the FPGA logic consists of an open-spec interconnect SoC bus called Wishbone [56]. Wishbone is an interconnection bus that makes SoC design re-use easy by offering a standard data exchange protocol. It supports many modern bus features such as interrupt vectors, user-defined tags, single-clock data transfers, and more. One of the most interesting features is

the multi-master capability of the bus, which is especially useful in this case since the goal was to connect the Linux processor and the real-time microcontroller on the same bus.

At first, it seemed like a good idea to connect the processor directly to the SoC bus on the FPGA. This is an approach that has been used by designers before [57]. It is possible to do this using the i.MX6Q's Extenal Interface Module (EIM) which is meant to interface with devices external to the chip. It provides both synchronous and asynchronous accesses to devices with SRAM-like interfaces. Since the EIM does not generate the exact signal patterns that are needed to control a Wishbone interface, it is necessary to translate the EIM bus transactions to Wishbone transactions so that it is compatible with the SoC bus on the FPGA. Bus transaction translation is necessary but can slow down the accesses and a performance bottleneck can be created as a result of the slower performance of the EIM compared to the SoC bus. A simulation of a simple read/write transaction from the i.MX6Q to the Wishbone bus is pictured in Figure 3.3. The first row of the waveforms is the EIM clock, and the FPGA clock is the second row. It is clear in the picture that the EIM takes longer to complete a transaction than the wishbone bus master.

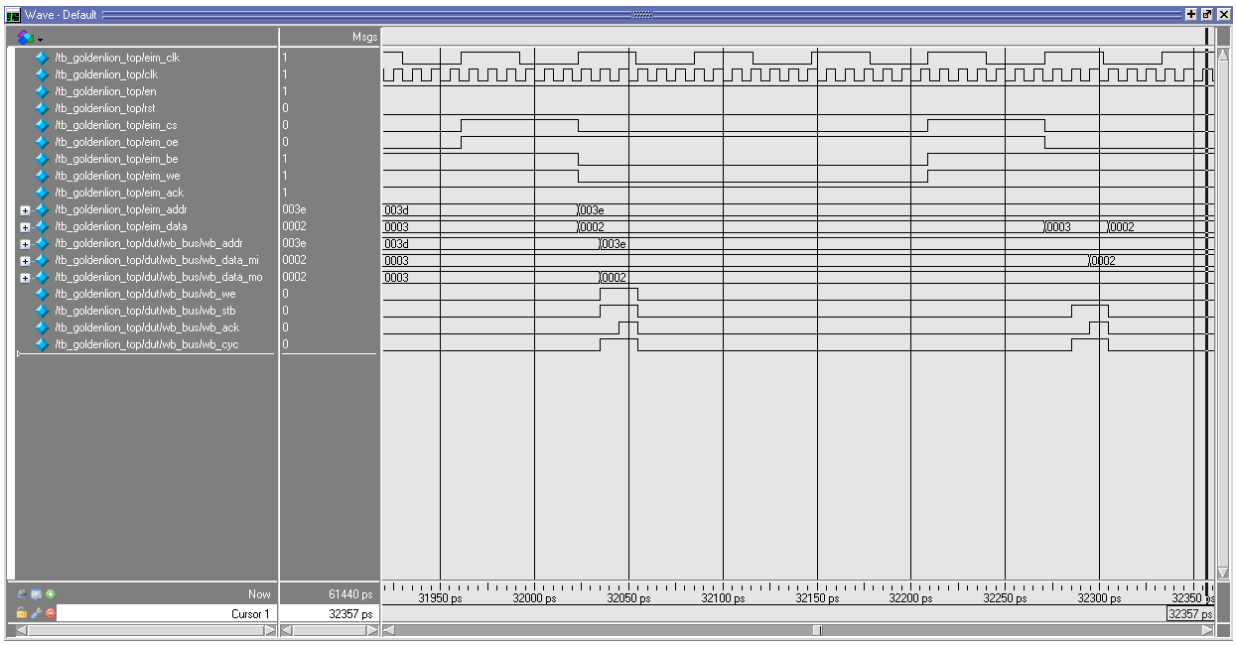

Fig. 3.3 EIM to Wishbone transaction

There are many issues that arise when using this sort of configuration. One issue is that it adds complexity to the internal bus design since the transaction converter is not trivial.

It is implemented as a state machine and contains six states. Another issue that arises is Cross-Domain Crossing (CDC). Although the simulation pictured in Fig. 3.3 does not take into account CDC, it can be a serious issue in designs where there are two ICs that communicate in an asynchronous manner. When taking into account CDC in a design, one has to consider the instance of metastability, which is the phenomenon in which signals do not assume stable 0 or 1 states for a certain time duration during operation. If such a signal in one domain crosses to the other, it can corrupt a good portion of the signals on the other logic portion with the different clock domain. There are known techniques to either dramatically decrease the chance of glitches or to simply remove the possibility altogether [58].

3.3.3 F4-Discovery Daughter Board

To prototype the functionality of the FSMC on the STM32F4, it was decided that a basic base board would be constructed. The name of this base board is the *F4-Discovery Daughter* board. In addition to prototyping memory-mapped communication from the microcontroller, it was also designeded for use in the Microprocessor Systems course (ECSE 426) to assisst in the education of McGill undergraduate students in the Computer Engineering program.

The circuit board was meant as a base board to an STM32F4-Discovery kit (pictured in Fig A.1). The reason this was done (as opposed to creating a circuit board with the microcontroller already placed) was because it was less complex to leverage an existing design than to do everything from scratch. The fact that the ST-Link programmer was integrated is also a nice addition since there is no need to use an external JTAG programmer.

The *F4-Discover Daughter* board contains many features geared towards education applications. One of the most important components is a gyroscope chip for firmware development with significant tilt angle awareness and for interacting with MEMS sensors. A USB-to-Serial FTDI chip will allow the microcontroller to output large amounts of data to a connected PC, without necessarily being in debug mode. It will also allow students to add features such as a user shell to increase user-based interactions. An on-board audio input jack is also installed so that a wide variety of signal and/or audio processing features can be explored. The addition of the audio input along with the MEMS microphone can be used to explore the processing power of the Cortex-M4s FPU and can lead to interesting

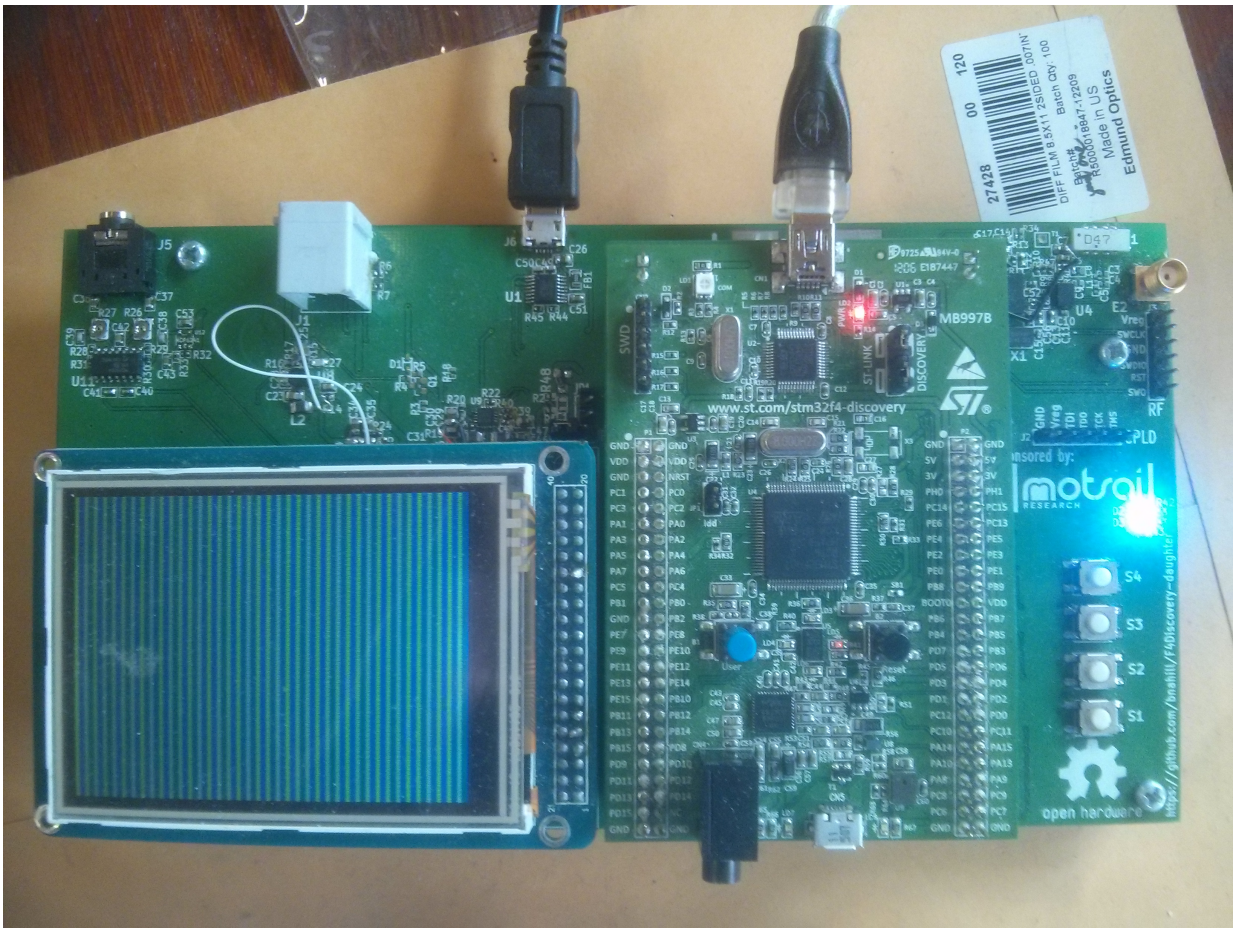

Fig. 3.4 F4-Discovery Board

embedded applications. The SD card slot that is also installed with the board will be ideal for storing and processing input samples.

A CPLD is used to manage most of the interrupt lines coming from all of the external peripherals. For example, instead of the accelerometer interrupt line being permanently assigned to a particular external interrupt line, it is reconfigurable to any of the microcontrollers external interrupt lines by the digital logic programmed in the CPLD. This allows for flexibility in terms of the peripherals used in various projects.

Last but not least, a Lego Mindstorms NXT sensor port has also been added for the purpose of assisting students learning about low-level drivers. This is probably one of the most unique features of this project. For a number of years, it has been used in Computer Engineering classes at McGill to introduce concepts in robotics, software and electronics.

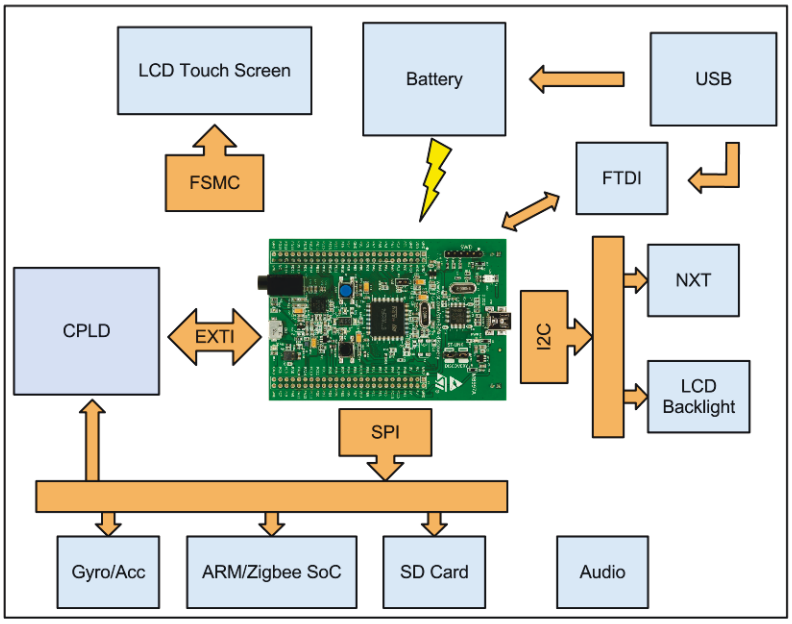

Fig. 3.5 F4-Discovery Daughter System Overview

Along with a reasonably powerful mobile computer, the NXT kit offers a wide variety of sensors and peripherals. So far, students have only interacted with these sensors from a high-level programming perspective. A simple function call was enough to get a sensor value. One of the objectives of adding an NXT port to this board is to reacquaint students with these familiar sensors that they used in their first year of schooling, but this time, expose them to the under-the-hood concepts of sensor communications.

3.3.4 FSMC Latency

The speed of the FSMC memory bus was calculated according to the clock tree schematic provided by the ST Microelectronics. The memory transactions are sent in an asynchronous manner, which means there is no clock signal involved in the transaction. The discovery board comes with an 8 MHz external crystal oscillator. The latency of the memory call is calculated as follows: This 8 MHz signal is directed to a phase-locked loop (PLL) that sets the signal speed to the maximum amount allowed at the system clock level, which is 168

MHz. The PLL sets its output frequency as

$$\frac{HSE * N}{M} \tag{3.1}$$

where N represents the multiplier and M represents the divisor of the respective input frequency HSE. The signal is then redirected to another divisor P before becoming the SYSCLK signal. To achieve the maximum system clock frequency allowed, the multipler M is set to 336 and the divisors M and P are set to 8 and 2, respectively.

$$\text{SYSCLK} = \frac{HSE * N}{M * P} = \frac{8 \text{ MHz} * 336}{8 * 2} = 168 \text{ MHz} \tag{3.2}$$

The SYSCLK signal propagates to the Advanced High-Performance Bus (AHB)'s clock without modification from prescalers or multipliers. This signal propagates to the FSMC. One clock cycle is defined as

$$HCLK = \frac{1}{168 \text{ MHz}} = 5.9 \text{ ns} \tag{3.3}$$

Since the data lines are used for both addressing and data, a lot of bandwidth is taken by the fact that the address of the transaction needs to be set up before transferring it. The speed of the FSMC is based on the configurations of the address setup and hold times as well as the data setup time. The amount of time it takes for a single 16-bit transaction is defined as

$$HCLK * (\text{T}_{asu} + (\text{T}_{dsu} + 1)) \tag{3.4}$$

where $HCLK$ is the time per clock cycle as calculated in (3.3), T_{asu} is the address setup time, and T_{dsu} is the data setup time, both in terms of the number of clock cycles. Therefore, this means that one 16-bit transaction should take $5.9 * (2 + (2 + 1)) = 29.5$ns.

To demonstrate the function of the FSMC and to verify the predicted latency, the FSMC pins that are connected to the LCD touch screen and are also connected to a 200 MHz logic analyzer. The signal trace is exported to a Modelsim compatible format and analyzed. A single memory transaction is shown in Figure 3.6. As predicted, the transaction takes approximately 30 ns if it is calculated as the time from when the NWE signal goes low for the first to the time when the signal goes back to high for the last time. However, this figure cannot be used to make specific assumptions about the bandwidth of the bus, since

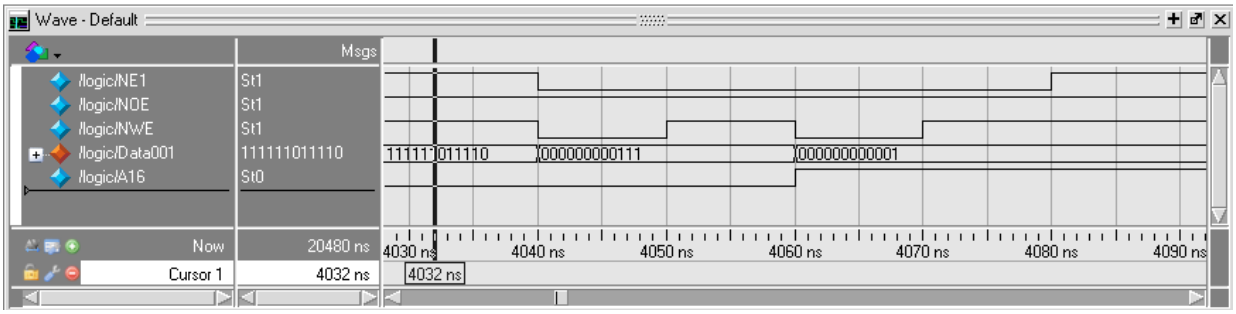

Fig. 3.6 Single 16-bit transaction

there is bus turnaround time in between transfers that must be taken into account.

3.3.5 Separate CPU and FPGA linked by PCI-Express

PCI-e is an increasingly popular protocol for internal data transmission with high-bandwidth requirements. As a result, more and more FPGAs come equipped with circuitry capable of performing PCI-e operations. These are called PCI-e 'hard' IP cores. FPGA manufacturers have been adding ready-made PCI-e functionality to make it easier for system designers to create co-processors accessible by memory-map. PCI-e can be used for either server, desktop or embedded applications. However, because of high power consumption of this bus, it is mostly used in desktops and servers. This bus was strongly considered for the communication between the i.MX6Q Linux processor and the FPGA due to the great potential for high-speed communications between the two chips. There was also the fact that it is relatively easy to set up on the FPGA side thanks to the hard-ip created by Altera. Another argument in favor of this option is that the i.MX6 contains a PCI-e peripheral with already-made drivers available. In the case of the FPGA family that was chosen for the *Golden Lion* project, the Cyclone V, the FPGAs came with a four-lane second generation PCI-e hard-ip core. Second generation PCI-e connections are rated as being as fast as 4 Gbit/s per lane [14] [10]. Since Cyclone V FPGAs can have up to 4 full-duplex lanes active, the maximum possible bandwidth for this PCI-e connection can be calculated as

$$\text{x1 lane} = 5.0 \text{ Gbps @ 8b/10b encoding} = 500 \text{ MB/s}$$

$$\text{x4 lanes} = 4 * 500 \text{ MB/s per direction} = 2 \text{ GB/s duplex}$$

A Modelsim simulation of the serial transfer in a of information from the PCI-e endpoint to an on-chip memory is pictured in Figure 3.7.

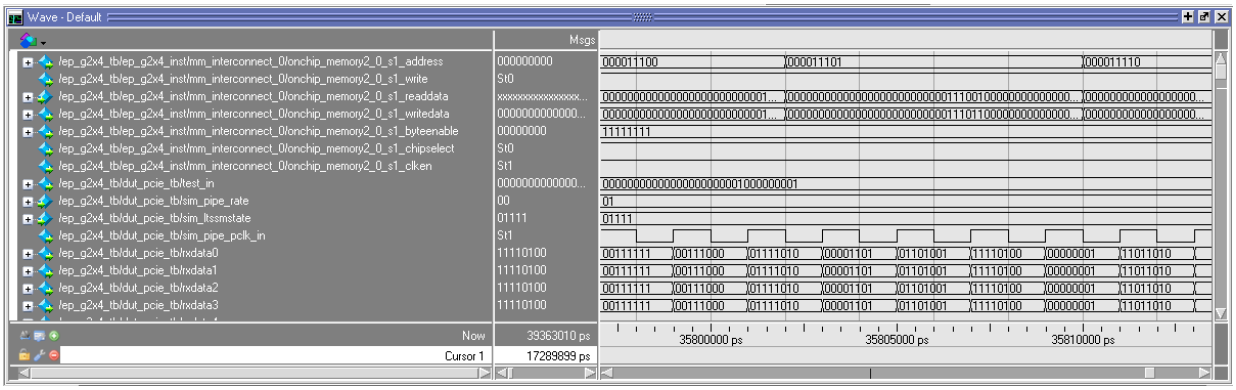

Fig. 3.7 PCI-e Transaction Burst Access

These numbers were very encouraging, but there is one big downside to this option, which is that it would have required laying out the high-speed serial connections on a printed circuit board (PCB), which is very difficult and would require expensive engineering design automation (EDA) tools that were not available. Because of this, it was decided that another format of architecture would be required for this system.

3.4 SoC FPGA based design

Designing a system based on a single SoC FPGA over a system that is based on decoupled parts has a clear advantages. The most obvious one is that all of the potential design difficulties related to integrating discrete components together are avoided. The other advantage is that the associated workflow for creating and testing the firmware, the register transfer logic (RTL), the linux kernel and the userspace programs is more straightforward. Integrating everything together is much easier when it is done on one chip. There are also significant cost savings associated with reducing board size and using less components. Altera's SoC FPGAs seemed like the most appealing option due to the availability of sophisticated tools [59] and sufficient amount of documentation and support. It contains an ARM-based Hard Processor System (HPS) consisting of an 800 MHz processor, peripherals, and memory interfaces with the FPGA fabric.

The circuit board used to develop experiments and prototype algorithms is the Arrow SoCKit development board, pictured in Figure 3.8.

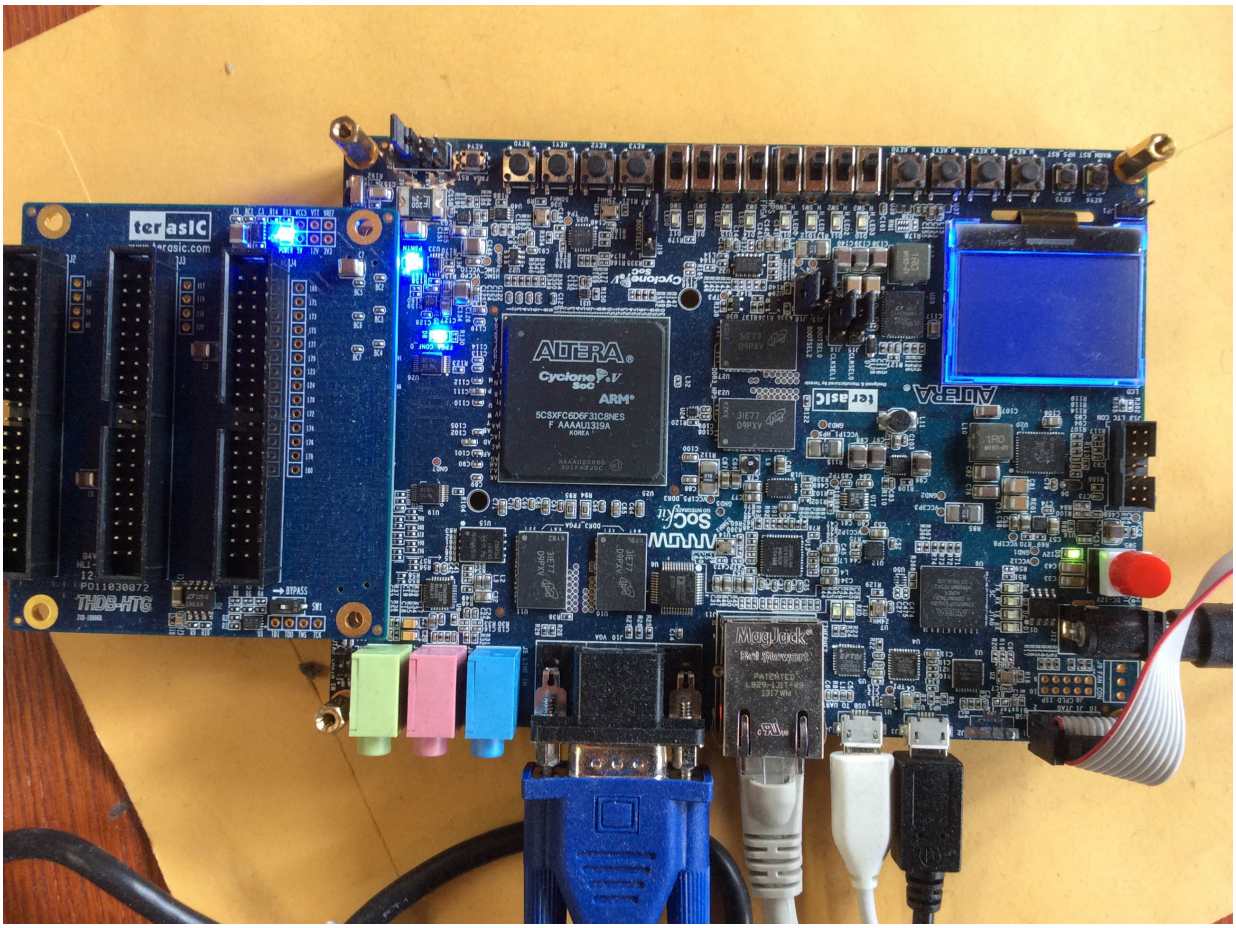

Fig. 3.8 Arrow SoCKit

3.4.1 Overall Architecture

The overall architecture based on a SoC FPGA is pictured in Figure 3.9. It retains the same overall arrangement as the previous proposals: a high-powered embedded processor running Linux, a real-time microcontroller and an FPGA fabric. However, the different components are more tightly coupled.

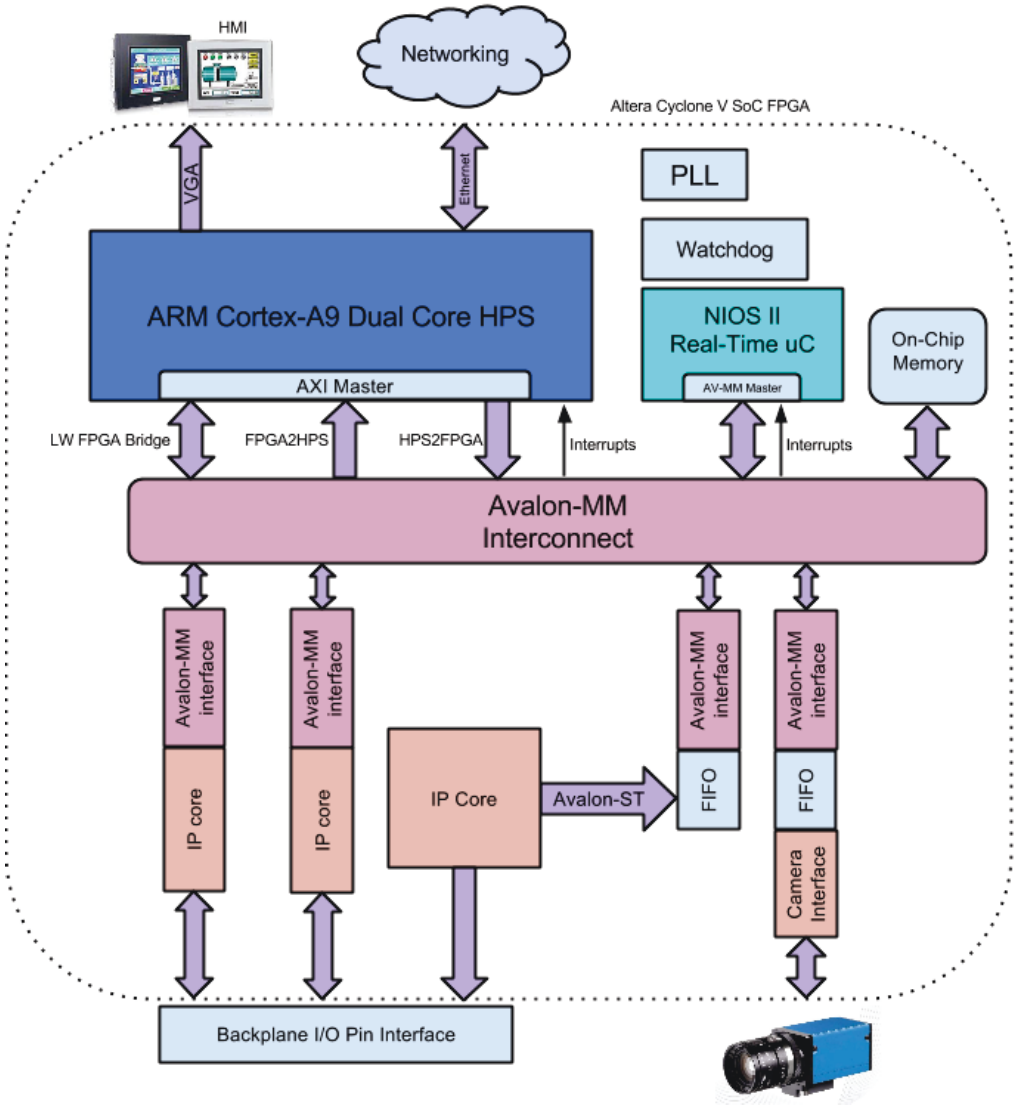

Fig. 3.9 SoC FPGA Overall Architecture

3.4.2 Hard Processor System

The ARM-based HPS present on the chip runs a Linux OS that does not have ”hard” real-time constraints. Its role is to handle HMI and networking functions as well as be the master of the rest of the system. This is also where the processor runs the API for MotSAI's IP cores. It is not as powerful as the i.MX6Q, which runs at a higher clock rate, has more cores, and contains more peripherals. However, the HPS is powerful enough to provide reasonable performance for HMI and networking applications. This family of FPGAs will

only get more powerful HPSs as technology improves. If there is a need to take on more powerful work loads in the future, the need could be fulfilled by more powerful embedded HPS modules in the future. The HPS logic communicates with the rest of the FPGA fabric through the means of three different bridges [60]. The FPGA-to-HPS bridge provides access to the peripherals and memory in the HPS. This is how a bus master on the FPGA would send data to the HPS or access any of its peripherals. The HPS-to-FPGA bridge provides a high-bandwidth master interface with an address space of 1 GB (minus 64 MB which is occupied by the lightweight bridge) to access any of the peripherals implemented on the FPGA. The lightweight bridge is similar to the HPS-to-FPGA bridge except the width of the accesses cannot be changed and the possible address map is smaller (64 MB).

3.4.3 Real-Time Microcontroller

Instead of having an off-chip real-time microcontroller like an STM32F4 that communicates by FSMC, it was decided to simply synthesize a *soft* processor and put it into the FPGA fabric. The NIOS II processor is one of the soft processors made available through the Altera development suite. This processor is capable of running at up to 160 MHz for the version that is free-of-charge and up to 200 MHz for the more sophisticated version [61]. oThis means it has reasonable performance compared to the maximum frequency of 168 MHz that the STM32F4 can run at. The microcontroller's instruction and data memories reside on an on-chip memory module. It is even possible to synthesize more than one instance of the NIOS on the FPGA and dedicate each one to their specific function(s). It is also possible to add custom instructions if needed. This processor was targeting the FreeRTOS real-time operating system.

3.4.4 System Interconnect

One of the most important features of this architecture is the unified system memory map. The NIOS real-time microcontroller addresses the peripherals in the rest of the system with the same address space as the HPS.

This provides exceptional flexibility for future developements and makes it easy to add new IP cores and adapt the system as they are developed. All of the components of the system present in the FPGA fabric are interconnected and communicate with each other through an internal SoC interconnect designed by Altera. The interconnect is a network-

	hps_0.h2f_lw_axi_master	nios2_qsys_0.data_master
hps_0.f2h_axi_slave		
led_pio.s1	0x0001_0040 - 0x0001_004f	0x0001_0040 - 0x0001_004f
dipsw_pio.s1	0x0001_0080 - 0x0001_008f	0x0001_0080 - 0x0001_008f
button_pio.s1	0x0001_00c0 - 0x0001_00cf	0x0001_00c0 - 0x0001_00cf
jtag_uart.avalon_jtag_slave	0x0002_0000 - 0x0002_0007	
intr_capturer_0.avalon_slave_0		
alt_vip_vfr_vga.avalon_slave	0x0000_0100 - 0x0000_017f	
onchip_memory2_0.s1		0x0003_0000 - 0x0003_ffff
nios2_qsys_0.jtag_debug_module		0x0000_1000 - 0x0000_17ff
jtag_uart_0.avalon_jtag_slave		0x0000_0028 - 0x0000_002f
timer_0.s1	0x0000_0000 - 0x0000_001f	0x0000_0000 - 0x0000_001f
sysid_qsys_0.control_slave	0x0000_0020 - 0x0000_0027	0x0000_0020 - 0x0000_0027

Fig. 3.10 Unified Address Space

based topology where transactions between masters and slaves are done with the use of packets [62]. It automatically manages clock-domain crossing and the transportation of transaction packets from the source to the destination. Modules communicate to this interconnect as either a master or a slave by initiating Avalon Memory-Mapped (AV-MM) transactions. These transactions are specified by the Avalon Bus specification and are interpreted by the interconnect to send it to the right place. Some IP cores that generate a lot of information can send that data through a streaming interface (Avalon-ST) to a FIFO buffer. These FIFO buffers are then accessible through the processor's memory-map. Interrupts generated by peripherals are also redirected throughout this interconnect to their respective recipients.

Flow Summary	
Flow Status	Successful - Sat Jun 14 15:01:58 2014
Quartus II 64-Bit Version	13.1.0 Build 162 10/23/2013 SJ Web Edition
Revision Name	soc_system
Top-level Entity Name	c5sx_soc
Family	Cyclone V
Device	5CSXFC6D6F31C8ES
Timing Models	Preliminary
Logic utilization (in ALMs)	6,781 / 41,910 (16 %)
Total registers	12440
Total pins	182 / 499 (36 %)
Total virtual pins	0
Total block memory bits	635,358 / 5,662,720 (11 %)
Total DSP Blocks	0 / 112 (0 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI TX Channels	0 / 9 (0 %)
Total PLLs	1 / 15 (7 %)
Total DLLs	1 / 4 (25 %)

Fig. 3.11 System synthesis report

The synthesis of a basic system as described above gives the following report. As can be seen in Figure 3.11, there is only a fraction of the available adaptive logic modules

(ALMs) that are used for the basis system, which leaves ample room for additional IP cores. Running the timing analyzer on the FPGA logic gives a maximum frequency of 70 MHz.

3.5 Operating System Configuration

The OS running on the HPS is Ubuntu Linux 12.04 running with a kernel 3.9. Linux is a monolithic kernel that includes the expected kernel components such as a scheduler, memory management, networking, an API, inter-process communication (IPC), and is mostly POSIX-compliant. Because it is monolithic, it includes device drivers, file systems, as well as a port for most existing computer architectures.

Most embedded system file systems are either taken pre-compiled from ARM distribution like Android, Debian or Ubuntu. If one chooses, the entire file system can also be compiled using the Yocto project. Compiling the kernel and all of the programs yourself has the advantage of providing more flexibility. However, it can become tedious in the long run to support everything yourself. For these reasons, it was decided that the *Golden Lion* project would be based on the Ubuntu distribution. The support from Canonical and the rest of the community maintainers will make future maintenance much easier.

3.5.1 Boot flow

The HPS boot starts when one of the processors is released from reset. It starts by executing the code that resides in the internal boot Read-Only Memory (ROM). The boot ROM performs minimal configuration and loads the preloader. Generally, the preloader can be found in multiple sources such as an SPI flash memory, the FPGA fabric, or from an SDMMC flash card. In this case, the preloader comes from an 32 GB SD card.

The HPS boots through the following stages:

1. Boot ROM
2. Preloader
3. U-Boot
4. Linux

The preloader is the program that configures the clocking, pinmuxing, pin I/O parameters (drive strength, logic levels, etc.), the RAM, and loads U-boot into the RAM. It is based on the secondary program loader (SPL), which is an open source bootloader. The source files for this bootloader is generated by an Altera program, which takes in the handoff files generated by the QSys tool [62] during synthesis. The flow for generating the bootloader is shown in Figure 3.12.

U-boot is the next major component of the boot flow. U-boot is an open-source bootloader with comprehensive support for managing and loading boot images. In this case, the image being booted is the Linux kernel. The program's main purpose is to manage the images and the rest of the hardware features before the boot of the OS.

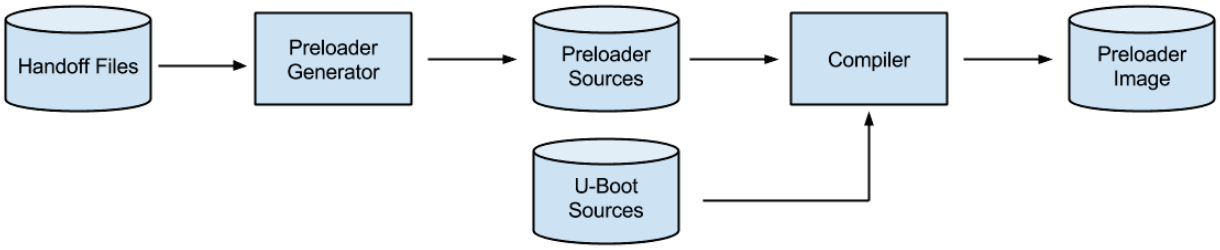

Fig. 3.12 Preloader Generation

The Linux kernel manages the rest of the boot up until the start of the intended application process. The *start_kernel()* function called from the bootloader performs the majority of the system startup, including interrupts, memory management and hardware device driver initializations. Once the kernel has started, the scheduler and the idle processes are the first processes to get spawned. In this case, the first process is *init*, which is the standard in the Ubuntu 12.04 LTS distribution. As a side note, *systemd* is starting to overtake *init* as the standard first process in Linux distributions such as Debian, Arch, Fedora and Ubuntu and will be seen more often in the future.

3.5.2 Device Tree Structure

The way to inform the kernel bootloading process of the available devices on the running hardware is to associate it with a device tree structure (DTS) file. The DTS is a structured file that contains the description of the running hardware and the available devices that is passed to the kernel at boot time. In the workflow generally associated with SoC FPGAs,

DTS files are among the handoff files generated during FPGA synthesis bitstream. The kernel does not interpret a plain DTS file, instead it interprets a compiled version of it called a device tree blob (DTB). When referred to a particular device in the DTS, the kernel loads the appropriate device driver to handle it. Once a device is registered, a running process has to pass through the kernel to interact with it.

3.5.3 Userspace I/O

Another way for the designer to interact with the hardware is to go through a memory-map function call on the */dev/mem* device, which is a device abstraction representation of the physical memory of the computer. One of the ways to use */dev/mem* to interact with hardware is to create a hardware driver program that runs in userspace and knows exactly which devices there are and where they are addressed. Memory-mapping */dev/mem* gives complete control of the entire physical memory address space to the program. Normally, a process running in user space can only access virtual memory pages granted to it by the OS, and physical memory is off-limits. The main motivation of this method is that it does not require the designer to inform the kernel of a device's existence to interact with it. However, there are a number of disadvantages to this approach. First, a program that interacts with */dev/mem* requires root permissions to function to do so. Second, the fact that the program has access to the entire physical memory makes it potentially very dangerous if it accesses unauthorized areas (areas reserved for the kernel, for example) and can take down an entire system. There is also no means of using interrupts.

This is where UIO comes in. UIO is a generic framework for handling devices in userspace [63]. Within this framework, device handling is split between a small part in the kernel space, which handles memory reservation and interrupt handling, and the part in the userspace that handles device functionality. It is a compromise between writing a full-fledged Linux driver, which can be time-consuming and difficult to debug, and directly accessing physical memory, which has its own advantages and disandvantages. The introduction of UIO as part of the Linux kernel was not unanimous, as many top contributors expressed reservations about its security [64] and the opening the door to situations where hardware vendors could circumvent the GPL license [65]. However, in the end, it was added as a permanent addition to the kernel.

Loading the UIO kernel module provides the structure necessary for individual user UIO

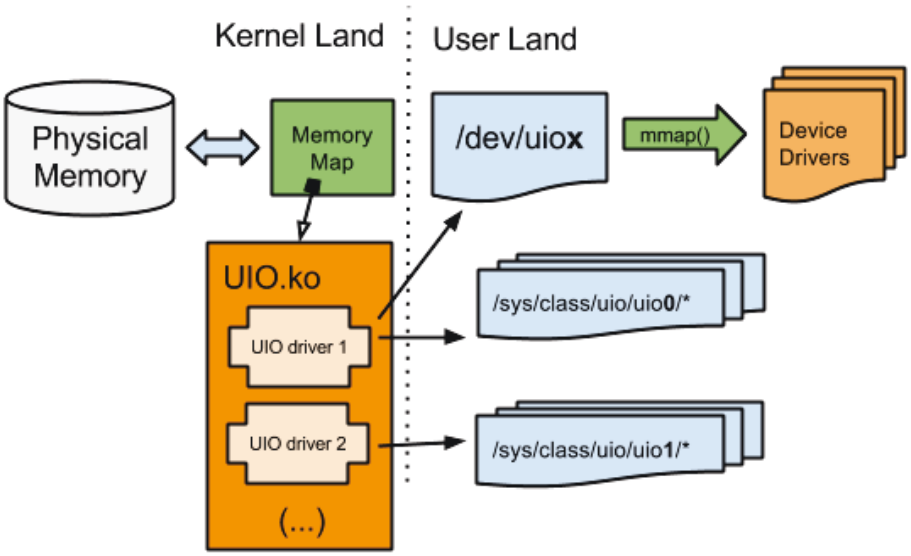

Fig. 3.13 Userspace I/O Framework

drivers to be loaded and executed. User UIO drivers are the ones that contain the minimum information about the memory space of the target device and provide a callback for the interrupts generated by that device. This allows for the program that actually interacts with the device to be written as a userspace program by performing an *mmap()* on the */dev/uiox* and */sys/class/uiox/** file handlers. Using UIO is great for rapid prototyping of drivers for devices residing in a programmable logic fabric. Such devices change often and the UIO framework offers the required flexibility that fully implemented drivers rarely provide. This is why it has been used to prototype and control many of *Golden Lion*'s core IP modules in the FPGA described in Chapter 4.

3.6 Dealing with evolvability

When designing an embedded system, a designer has to increasingly take into account the capacity for that system to adapt to a change in requirements and part obsolescence. In other words, embedded systems have to be designed with evolvability in mind [1]. Basing the design of the *Golden Lion* project on SoC FPGA technology brings about significant advantages in this regard.

Deploying an embedded system based on an FPGA means that there is a good chunk

of the IP developed for that system that is RTL. Since RTL is portable to other FPGAs, in the event that the FPGA malfunctions and has to be changed, the same IP core can easily be ported to a new family. This is similar to approaches suggested by publications in this field of research [4] that strongly suggest FPGAs as the basis of the design for industrial systems.

3.6.1 Software Evolvability

Porting firmware to a new platform is a different story. Being able to simply move a compiled binary blob from one platform to another is very rare. By the time obsolescence does actually happen, it is very likely that a generation or more's difference in processor architecture will make that firmware blob incompatible. A synthesizable "soft" processor like the NIOS II, which is used as the real-time microcontroller in *Golden Lion*, can help avoid some of these issues. In the event of obsolescence, a soft processor can simply be transferred to a new FPGA along with its firmware and avoid any new issues. However, there is a caveat, since the synthesizable processor being used is strongly associated with the FPGA manufacturer and cannot be used in an FPGA manufactured by a company like Xilinx since it is an encrypted IP core. Because of this, it is probable that *Golden Lion* will be based Altera's FPGAs in the long run.

Porting firmware to different architectures is a situation that Hallmans et al. present as a case study [6], where a legacy system based on a decoupled chip and FPGA can be upgraded to a system based on a Xilinx SoC FPGA that contains an ARM Cortex-A9 processor.s This is a case that is strongly considered in the design of this platform, which is why a SoC FPGA based design has been chosen. The only difference is that their use case considers replacing a CPU architecture with a different one. The approach in the case of *Golden Lion* is to keep the same soft-processor across multiple generations of FPGA families.

Choosing Linux as the OS for the *high-level* processing portion of the system is one of the best ways to deal with evolvability. Dealing with legacy Linux applications is not too difficult, thanks to the fundamentally open-source nature of the OS. Because it is open-source, it has the flexibility to adapt to new requirements. It also relies on the efforts of a large community of developers who are constantly fixing bugs and developing new features. The fact that there are no license fees or royalties and that there are multiple providers of

software means that it is the more economical choice as well as less expensive to improve and modify in the long run.

3.6.2 Evolvability of requirements

As a design evolves over time, different functions are added or removed depending on changes in requirements. Hallmans et al. have developed a novel method for gauging the effect of adding or removing features for the whole system [7]. This method is based on drawing a dependency graph between different features of a system. There is a set of dependency requirements for each function F that depend on set of communication (C) and I/O modules (IO). Hardware platforms are represented by H.

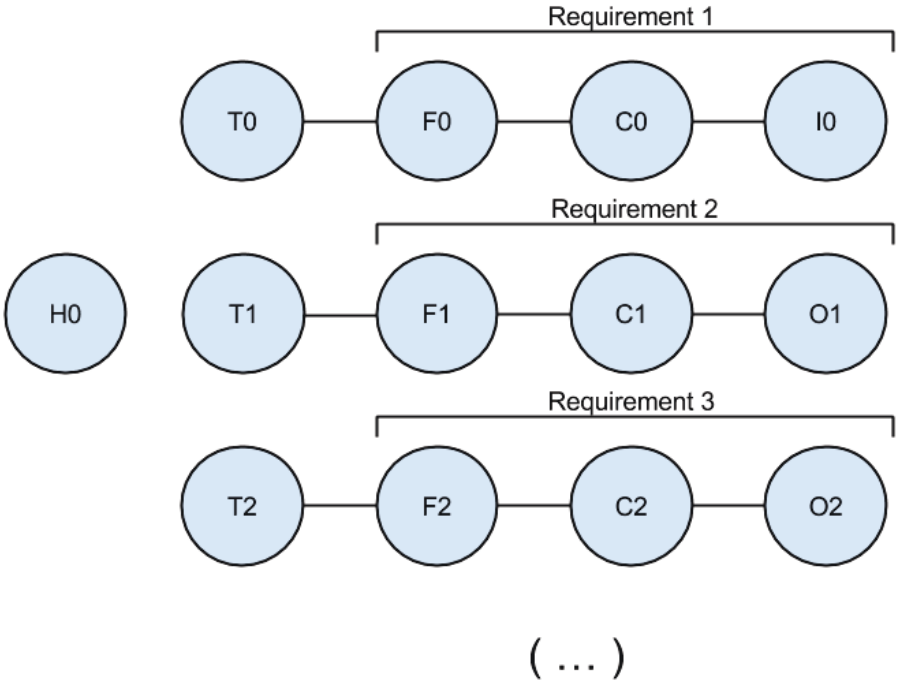

Fig. 3.14 Golden Lion dependency graph

A dependency graph based on this method has been drawn to show the evolvability of the platform over time and is pictured in Figure 3.14. Because the core of the overall system is modular, all of the functional requirements have no co-dependency and are easily interchangeable. Removing or adding a function does not depend on the rest of the system and is not a complicated process.

3.7 Comparison with other works

The architecture of this system resembles other system architectures that use SoC FPGAs as their core. Since FPGA vendors supply most of the digital structures that interconnect different IP cores, most designers are going to use these pre-designed structures for their projects. The end result is a SoC FPGA-based system that uses the provided interconnects and its accompanying memory-map mechanisms very similar to the one in *Golden Lion*. Even with Xilinx SoC FPGAs, the end result can be very similar, as is seen in this proposed system design that renders images of the Mandelbrot set [66]. Researchers have also used this form of architecture for video encoding [32]. Their designs are very similar to *Golden Lion* because it relies on on-chip peripheral busses for communication between the main processors and the co-processors.

Many commercial products take advantage of SoC FPGA. A notable one is HIL test simulators designed by Opal-RT technologies [67]. Their architecture is similar to the prototype described in section 3.3.5, in the sense that it uses a high-powered CPU connected to an FPGA co-processor by PCI-e. The FPGA fabric is used to generate the high-frequency signals communicating with external ECUs and external busses. The difference is that their high-powered CPU is intel-based, while the one described in section 3.3.5 is an ARM quad-core. They also don't use a real-time microcontroller.

The way the co-processor and the main processor interact is less coupled than some implementations [40]. Hodjat et al. use a custom LEON core instead of a hard ARM core and use special instructions instead of a memory-map to communicate with the co-processor. Although they find that using a custom instruction increases performance, it would be difficult to replicate this approach with an ARM core that is not modifiable. However, it is possible to use this approach with one of the "soft" NIOS processors by adding custom instructions.

There exist system designs based on SoC FPGAs that use automated software/hardware co-design compilers [68]. These compilers compile code that leverage the presence of both processors and the programmable logic fabric. The disadvantage of this approach is that it can be difficult to make sure that the system behaves as intended. It is also difficult to deal with evolvability, as these compilers are mostly proprietary today and may not work very well with different families of FPGAs. For this reason, that approach was not used in our project.

Chapter 4

Design Details

In this chapter, the design details and performance analysis of the components of the development platform are described. The design of individual IP cores are shown as a case study in the use of the *Golden Lion* platform. They were chosen in accordance to the context of programmable controller in an industrial environment as well as basic components that should run on FPGAs. The details in the implementation are also explained and compared to architectures illustrated in the literature.

4.1 Case Study: PID Controller

The Proportional-Integral-Derivative (PID) controller is one of the most widely used feedback-based control algorithms in the field of industrial control systems. Normally, this type of algorithm is executed on a microcontroller and can perform relatively well in most situations. In this case, it was decided to implement the algorithm on an FPGA for several reasons. First, the concept of redundancy and reliability is important in the context of industrial applications. With FPGAs, placing several PID modules in a single unit is not difficult and a malfunctioning CPU will not necessarily cause it to crash. Moreover, adding more PID cores alongside it will not affect the behaviour of the individual cores because of the ability of FPGAs to execute algorithms completely independently and in parallel. Consequently, it was deemed fitting that this would be one of the example uses of the *Golden Lion* platform.

4.1.1 Implementation

The implementation of the PID block is based on the standard control algorithm that produces an output $u(t)$ based on the control input $r(t)$ and the behavior of the plant $y(t)$. It is adjustable by tweaking the gain values *Kp*, *Ki* and *Kd*. This particular PID block used in *Golden Lion* is one that was designed in-house by Dr. Omid Sarbishei at MotSAI Research. It was designed and simulated using Scilab [48].

The IP block is wrapped around a control interface which consists of a set of registers that are accessible by the CPU through memory accesses, as can be seen in Figure 4.1. The CPU can change the PID's *Kp*, *Ki* and *Kd* during operation as well as its modes of operation. The accesses to this IP core block come through the Avalon-MM interconnect as described in Section 3.4.4.

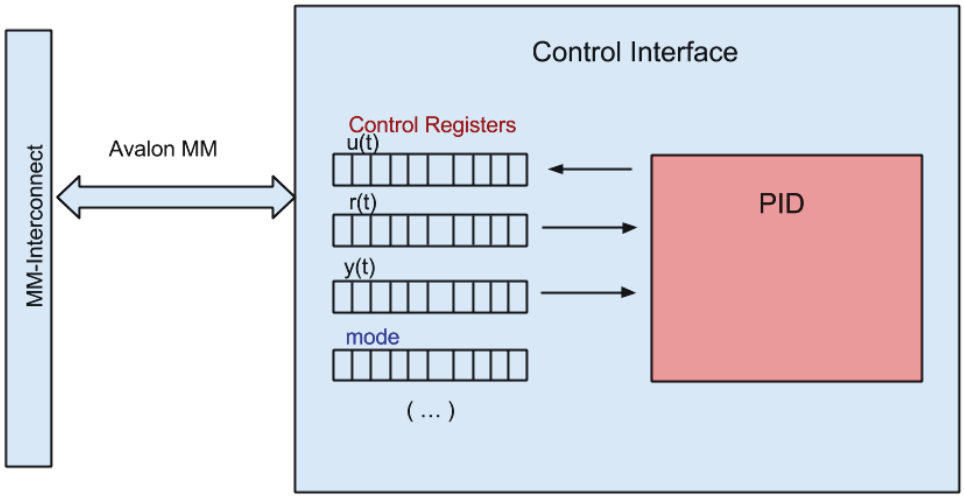

Fig. 4.1 PID Interface Module

4.1.2 Simulation

Normally, a controller produces a control output sample based on the feedback of the plant on each pulse of the clock signal fed that is fed into it. In the case where the controller is being validated, it requires that the PID produce an output sample when the CPU tells it to. This requires the control interface of the PID to have a feature where the controller is in *one-shot* mode and evaluates one sample only when explicitly signaled. The *real-time* (default) mode is the mode where it produces an output at each clock pulse. The mode can

be controlled by one of the memory-mapped control registers. The verification program on the CPU side runs as a Python script. Accesses to the FPGA is done through a C module which contains the logic of interacting with the components in the FPGA through */dev/mem*.

To verify the proper functioning of the PID block, it requires the CPU to feed the controller sample input values as well as the feedback of a simulated plant. The test vectors are comprised of $r(t)$, $y(t)$ and the expected feedback of the controller $u(t)$. To obtain a step response from the controller, the input test vector $r(t)$ is a step input. The FPGA's step response is pictured in Figure 4.2.

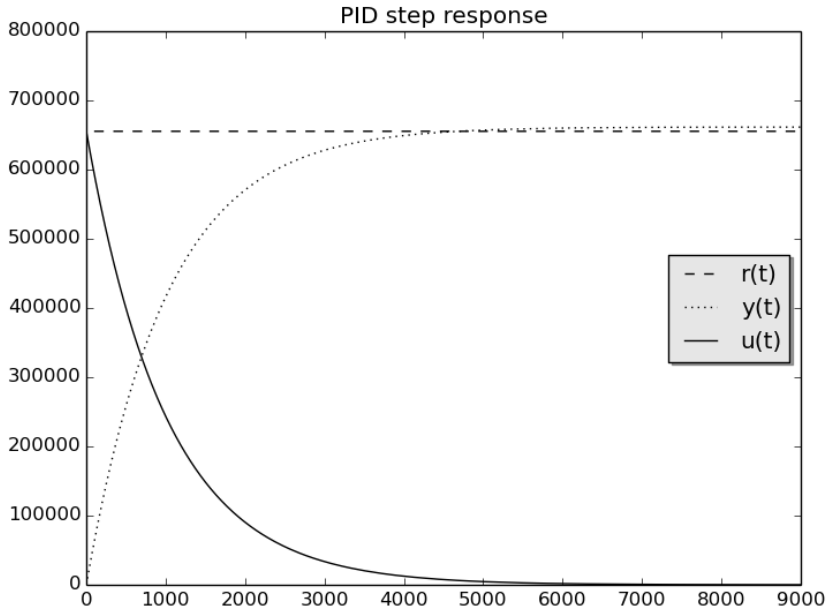

Fig. 4.2 Hardware PID Step Response

By saving the returned input values from the PID and comparing them to the ones obtained in the Scilab simulation, the CPU can make sure that the PID is functioning correctly while programmed in the FPGA. This is a form of HIL testing since the plant is being simulated while the controller is actually is being executed. The plant output, reference and controller output signals are all precomputed through simulation and are loaded into memory for the duration of the validation.

4.2 Case Study: Numerically Controlled Oscillator

A Numerically Controlled Oscillator (NCO) is a digital signal generator which creates a discrete-time and discrete-valued sinusoidal waveform. NCOs are widely used in industrial applications such as control and communication systems. They are usually used in conjunction with a Digital-to-Analog Converter (DAC) to create a Direct Digital Synthesizer (DDS). Among the obligatory sine and cosine waveforms, this particular NCO was also designed to generate a sawtooth and square signals. This IP core is implemented as a component that can be connected to the rest of the system from the beginning. Similar to the PID detailed earlier, all of the control and status registers are accessible by memory map through an Avalon interface.

4.2.1 Implementation

At the core of the design is a continuously incrementing phase accumulator. The phase of all the generated waveforms is defined by this register. To generate the sine and cosine waveforms, the accumulated phase signal is fed into an on-chip dual-port ROM that acts as an Look-up Table (LUT) for sinusoidal waveforms. Some implementations use two separate LUTs for the sine and cosine waveforms. In this, it was decided to use only one LUT for both by creating the cosine phase signal by adding $\frac{\pi}{2}$ radians to the sine phase signal. This was done to reduce on-chip memory usage as much as possible. All of the waveforms except the square wave (which takes only one bit) have 12 bits of resolution to limit the amount of on-chip memory taken up by the LUT. The outputs of the waveforms are of unsigned format, which means that a 1 is added to the output of the sine and cosine functions to make the range always positive.

The waveform contained in the LUT is generated using the Python code in B.1. The code takes in the data width as well as the address width and produces a list of output values in a text file. The synthesizer uses this text file to fill the contents of the LUT before operation. Its contents are described by the equation 4.1:

$$\mathrm{LUT}(x) = S * (1 + \sin(\frac{2\pi x}{L})) \tag{4.1}$$

where x is the address signal being fed into the ROM, L is the length of the LUT in terms

of number of samples and S is the scaling factor calculated according to the data width:

$$S = 2^{B-1} - 1 \tag{4.2}$$

Multiplying the sine wave by the scaling factor S is meant to make the sine waveform take the full possible range of the data width B.

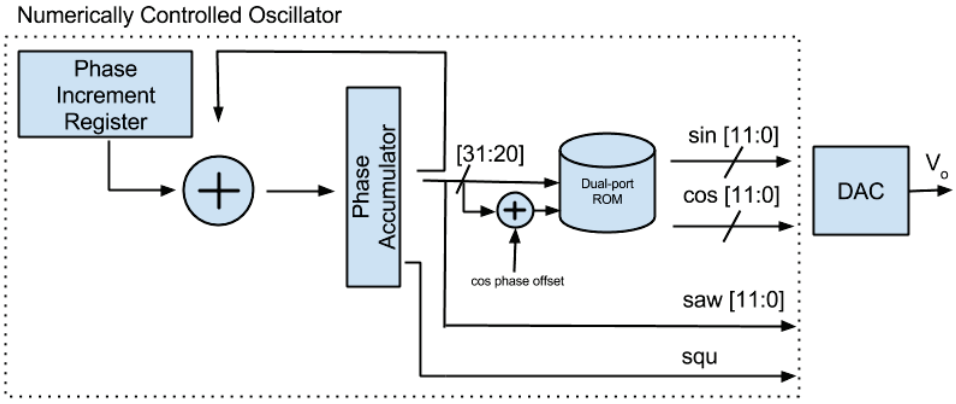

Fig. 4.3 NCO Hardware Structure

The oscillation frequency of the NCO is controlled by how much the phase is incremented at each clock cycle. The *phase increment* register contains the exact value in radians that is incremented and represents the discrete frequency at which the oscillator is operating. Since the 12 most significant bits (MSBs) of the *phase accumulation* register are used as in the input into the LUT, only an increment of the 21^{st} bit will have an effect on the output waveform. The oscillating frequency is determined by:

$$F_{nco} = \frac{F_{clk}}{2^{32-B_a}} \tag{4.3}$$

where F_{clk} is the frequency of the input clock and B_a is the amount of address bits (which is also the width of the phase accumulation register). A diagram of the structure of the NCO is pictured in Fig. 4.3. It is important to note that due to the digital nature of the circuit, there can be distortions that appear in the output waveform. A phenomenon known as phase truncation may occur when truncating the phase accumulation register. Phase truncation introduces non-harmonic distortion proportional to the number of bits being truncated. This can become a problem when dealing with frequency shifts with many

decimal points or frequency shifts represented by non-rational numbers such as multiples of π or $\sqrt{2}$.

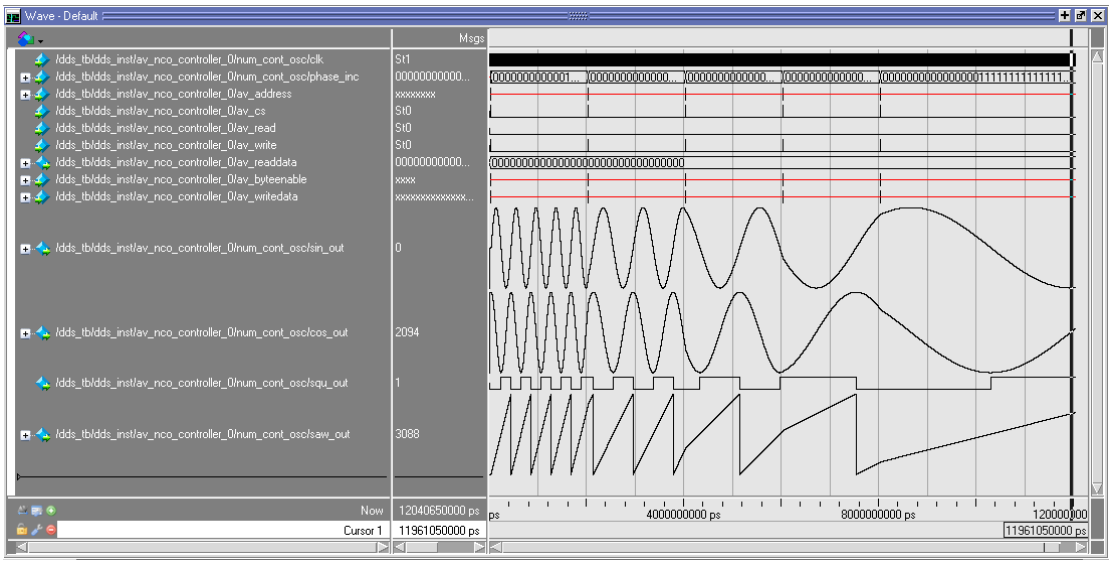

Fig. 4.4 NCO Implementation

4.2.2 Simulation

To verify the proper functioning of the NCO as well as its component Avalon control interface, a testbench consisting of an Avalon bus functional model (BFM) was designed. The BFM itself is produced by Altera. A simple use case in which the CPU changes the frequency of the sinusoidal waves as well as the sawtooth and square waves is pictured in Figure 4.4. One can easily notice that the frequency of all the waveforms change at the same time. This happens because all of the waveforms are tied to the same phase values. Possible improvements could include being able to control the frequency of the different waveforms separately. It would also be valuable to MotSAI Research to build a proper DDS based on this NCO to execute complex control algorithms.

4.2.3 Potential Applications

Somce it is paired with a real-time microcontroller, the NCO can perform various useful Digital Signal Processing (DSP) applications. Because both the phase and frequency are accessible and modifiable by an external processor of both the *sin* and *cos* functions,

analog modulation schemes such as Phase Modulation (PM) and Frequency Modulation (FM) are possible. Digital modulation schemes such as Frequency Shift Keying (FSK), Multiple Frequency Shift Keying (MFSK), and PM are also possible. The logical step from implementing these encoding schemes is to provide an interface for the GNU radio project [69] which is a great way to provide visibility to the platform.

One of the NCO's main intended applications is for industrial machines with large servo motors. By pairing the NCO with a feedback control system, it can constitute a key component of a DDS system. A DDS is ideal for driving motors that have sinusoidal inputs and where the precise control of the rotation of a motor is needed.

4.3 Case Study: LMS Adaptive Filter

In industrial design, filters are used quite often for many application including signal processing and control. As an important example of what is possible with the *Golden Lion* platform, a special type of LMS adaptive filter was designed. Although a software implementation of the algorithm is quite viable for most situations, a hardware implementation is still necessary for some high-speed applications. In fact, most filtering algorithms can be implemented very well on a FPGA. The following section explains the design process for an adaptive filter that is simulated and later implemented in an FPGA.

4.3.1 Simulation

When designing an adaptive filter, its convergence rate is important in the situation where a training signal is being used for calibration. There are some cases where an adaptive filters convergence rate might not be fast enough for high-speed data transmission. A slow convergence rate increases the proportion of the training signal in the total transmission, which has the consequence of reducing the overall effective bandwidth. This is an example use case where the implementation in an FPGA is ideal.

To start off the implementation, a golden software model was made so that the design could be verified to be correct. A script in MATLAB simulated the algorithm using floating point representation on all of the signals. This was done so as to concentrate on designing the algorithm itself and not get bogged down on the implementation details of fixed-point arithmetic. The simulation itself consisted of an LMS adaptive filter that attempts to converge to a given FIR filter. The FIR had an order of 140 coefficients and was designed

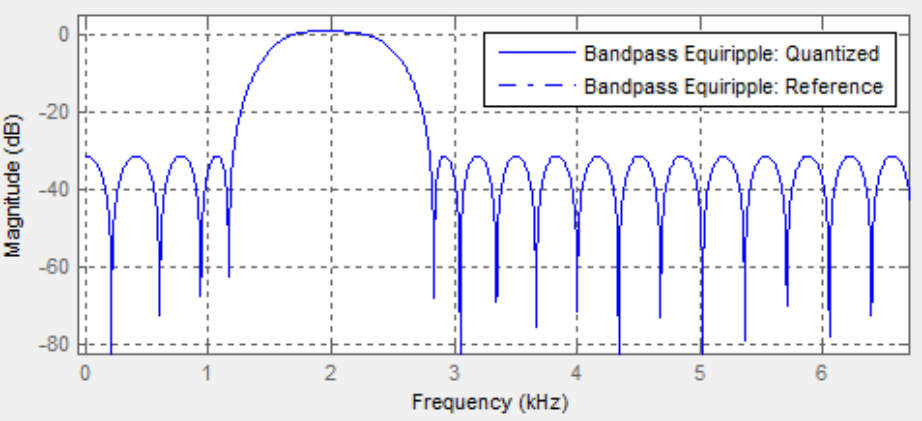

Fig. 4.5 Magnitude Response of reference filter

to be a band pass filter with pass band going from F_{pass} = 1.6 KHz to F_{stop} = 2.4 KHz with a magnitude response pictured in Figure 4.5.

The model transmission signal x[n] being fed into both filters is a sine wave of 2.0 KHz integrated with noise of variance of 0.5. The adaptive filter starts out with its filter weights at 0 but converges to the values of the reference FIR as you feed it more and more values. In the simulation, it takes as much as 8000 samples for the taps to converge to the right values. The variance in the noise and gain had to be tweaked carefully in order to have a reasonable convergence rate and to be able to converge to the right values. Fig. 4.6 shows the convergence of the filter to the appropriate filter values. The coefficients are pictured on the y-axis and time is on the x-axis. The values converge to the ones in the ideal FIR filter.

4.3.2 Implementation

During the design of this adaptive filter, an emphasis was placed on resource and area usage. To reduce area usage a serial architecture was chosen. The serial design that was devised is one that was described by a publication by Bernocchi et al. [70] but without the RNS arithmetic. The authors describe the circuit as updating the filter weights one at a time to be able to only have one MAC unit for the weights and one MAC unit for the data path. The implementation presented by this paper is what is used for this module. The final implementation takes up a lot more control modules for the data path so that everything syncs up properly. The entire block diagram is shown in Fig. 4.7. A state machine controls

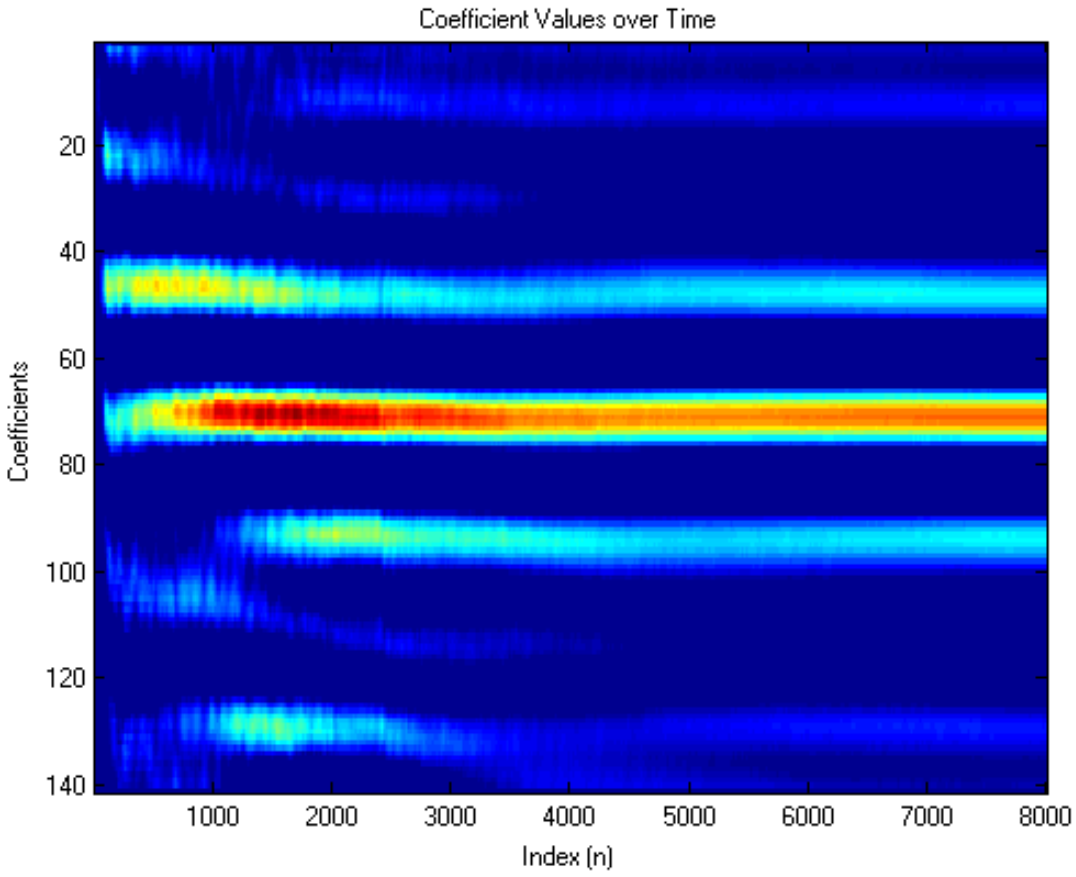

Fig. 4.6 Coefficient Values of Adaptive Filter

the multiplexer's select lines to be able to coordinate the data path properly. The algorithm emulated is described in a series of steps:

1. Shift in the input
2. Compute the output sample by multiplying each weight with the appropriate pipeline value using the Samples MAC Unit
3. Once a sample has been computed, recalculate all the weights individually using the computed error sample error(n) = desired(n) − y(n).
4. Once all the weights have been computed go back to step 1

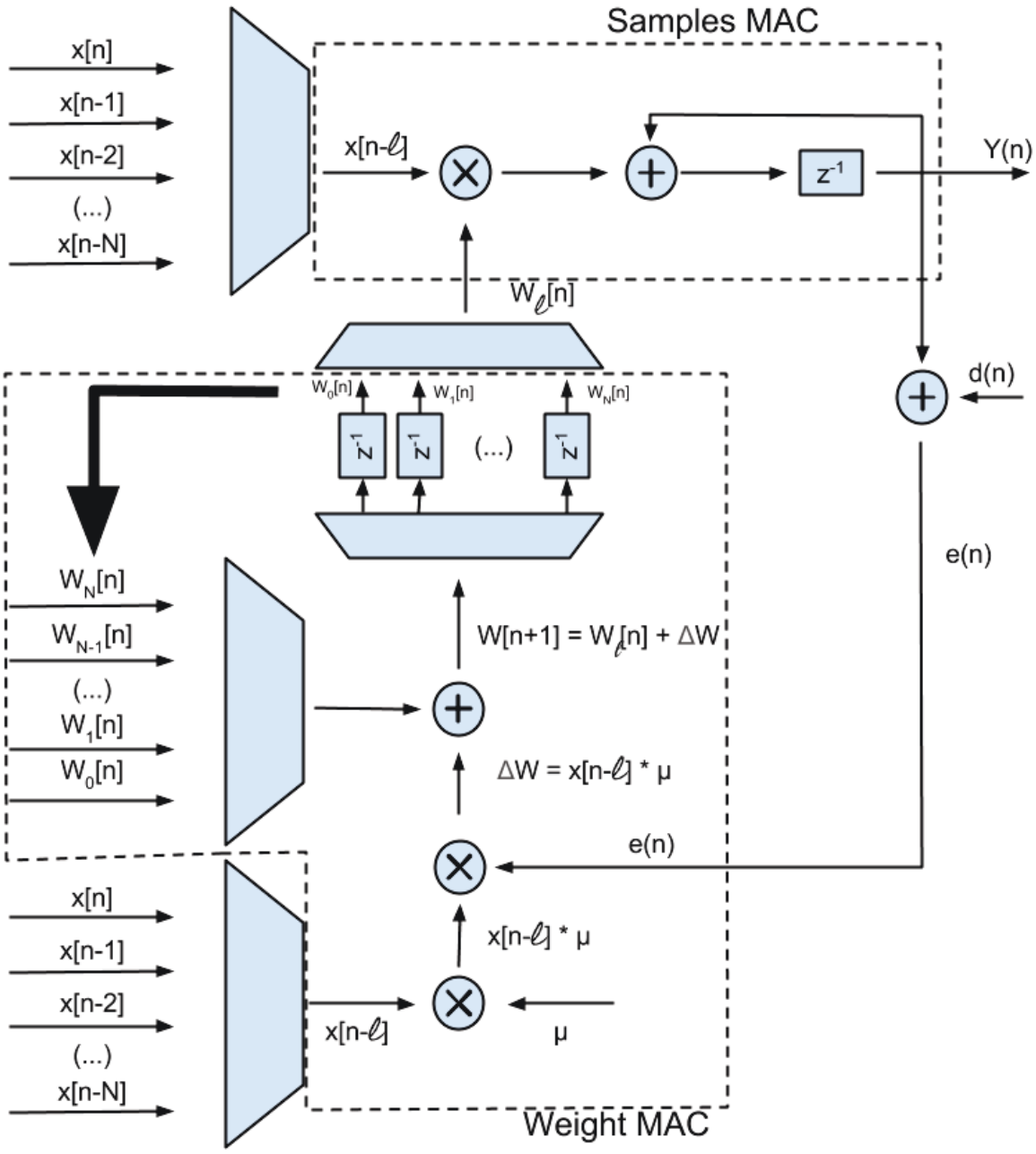

Fig. 4.7 Serial Adaptive Filter

The weights should converge to the right value after a few thousand samples. The most unique aspect of this design is the fact that there are only two MAC units for the entire adaptive filter. There is the Samples MAC Unit which is a standard MAC and there is

the Weight MAC unit which is pictured in Fig.5. The only difference between the weight MAC unit and a standard one is the fact that there are multiple values that accumulate every cycle, instead of the same value. This should result in big savings in terms of total amount of multipliers. This is an important conclusion if the amount of FPGA multipliers is limited. However, a register array will still be needed in order to retain the weights in memory.

Simulation

To compare the convergence rates and resource utilizations of both types of filters, both were implemented. A standard FIR filter was designed and contained the same order as the one that was implemented in MATLAB. The signals being fed into the filters through File I/O are generated in MATLAB. Doing a test of the standard implementation of the adaptive filter was straight-forward because it was being fed the same clock rate as the reference FIR reference filter. However, the tricky part is the fact that it takes 2N clock cycles to create a sample as opposed to the FIR filter that creates a sample every clock pulse. This meant that the clock that was fed into the reference FIR filter had to be divided so as to be in sync with the reference FIR.

Chapter 5

Development Environment

This section describes the use of various development tools to create programs that run on the Golden Lion platform. It will explore the use of programming languages amd simulation software and debugging tools built to assist engineers that use this platform for their products and experiments. This chapter contains the main academic contributions of this thesis, which is to demonstrate and analyze the use of a HIL simulation testbench with SoC FPGAs and the design of a tool that helps debug memory-mapped peripherals.

5.1 API Languages

The Golden Lion project is essentially a collection of intellectual property modules that run on modern hardware. To allow purchasers of this platform to use the IP, there is a need to give access to it from developers' programs. Since it is not possible to expose the IP to every programming language out there, a select few have been chosen for their strengths and weaknesses that do not overlap.

The first language chosen for an exposed API is C. C is a language that has been in use since the 1980s. Its main strength is its speed of execution. Providing an API for this language is mainly for power users that want to get all of the extra ounces of computing power they can get from the hardware.

The second language is Python. Python has the advantage of being a language that is easy to read, write, and understand. The semantics are simple and the programmer can get some reasonable execution speed out of it as long as he or she does not attempt too much parallel processing with it. An example hook that was built in Python is the one for the

PID controller implemented on the FPGA. With this Python file *pid.py* (in Appendix B.2), a programmer is able to verify that the digital module is functioning properly, start, stop, and is able to adjust the parameters. All of the communication is done through the CPU's memory map. Some interesting experiments can be performed with it without having to delve too far into the complications of messing around with the FPGA itself. It can also be used for interacting with peripherals through UIO, similarly to how it was done in an application note written by Jean-Samuel Chenard [46] and described in section 3.5.3.

5.2 Scilab and Xcos

After careful research and experimentation, the Scilab simulation software has been chosen due to the availability of documentation and its flexibility [48]. Another reason is Scilabs ability to simulate both discrete and continuous-time models in the same simulation environment without any difficulty. Moreover, the software kit Xcos which is included in Scilab allows for easy development of simulation models. This was a big motivating factor in choosing this software. Xcos is very similar in feature sets to existing commercial products. However, unlike most commercial products, Xcos is open-source. Being open-source is a great advantage because it gives access to the internal workings of the software and allows for modification of the software if necessary. It also has a plugin so that Modelica [71] blocks can be used. Modelica is a non-proprietary language which models complex physical systems containing electrical, electronic and other subcomponents. Using modelica blocks can decrease the development time if a user wishes to test equipment with complex physical models. The availability of custom C hooks from inside Scilab simulations is a useful tool for experimenting with various mediums for outside communication.

The Ptolemy II simulation software [72] was also considered due to its quite extensive toolbox and its ability to create custom blocks in the form of actors. However, the only downside was that the simulation itself runs on a Java Virtual Machine. Any calls to the native memory are required to go through the Java Native Interface (JNI). An experiment has been performed to compare the latency of writing to physical memory from both native C and from the Java Virtual Machine. It consists of both implementations which perform thousands of consecutive memory writes. The time it takes to perform these writes is recorded and compared. The results of these tests can be seen in Figure 5.1.

The tests show that using the Java Native Interface (JNI) for memory transactions to

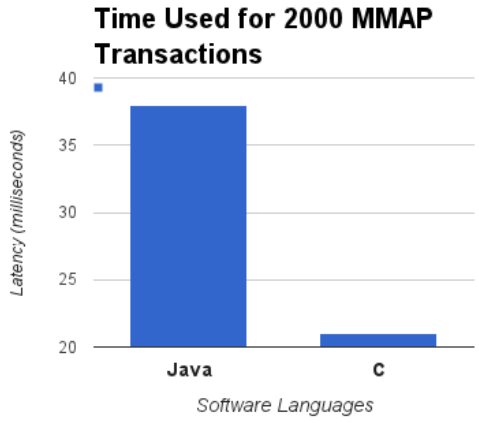

Fig. 5.1 Comparison of time used for 2000 MMAP memory transactions.

physical memory can have a significant impact on performance due to the Java Virtual Machine (JVM)'s overhead. As a result, the option of using the Ptolemy II simulation software is abandoned. Although Xcos also uses Java for the UI, the actual exchanges to the hardware are made in C thanks to the custom hooks available in Scilab.

5.3 Hardware-in-the-Loop modelling with Scilab on a Desktop PC

5.3.1 Motivation

When designing modern industrial PLCs, there is a need to be able to test control algorithms without necessarily having access to industrial machinery, which these algorithms control. This need can be resolved by simulating a plant on synchronous modelling software that is being executed on a PC. This form of HIL allows for quicker validation of controller platforms. Although this is a feature that is present in proprietary software systems such as MATLAB Simulink [47] and National Instruments LabVIEW, it is desirable to introduce this feature in existing open-source modelling software so that it can be used without having to deal with licensing costs and to allow for a more flexible experimental platform and for direct product derivatives to be built. Ultimately, this form of modelling will be available to developers using the *Golden Lion* platform.

5.3.2 Experimental Setup

This section describes the setup and results of an experiment where synchronous simulation software is run on a PC. In this particular case, it is desirable to link the discrete time model to interact with actual external hardware. One of the key issues in synchronous computation is the fact that simulation execution time depends on the convergence time of continuous models as well as the update time of discrete models. The latency of a system call to the hardware and how much effect it has on the real (physical) simulation time and the resulting system-level constraints are studied. The external hardware is run in a closed loop simulation with continuous time software running on a Linux system, which forms a proper HIL system.

Two communication mediums were considered for connecting a PC to the external hardware: USB and Ethernet. Peripheral access through USB allows for a middle ground for moderate latency and bandwidth measures. For the sake of software simplicity running on both the PC and the microcontroller, a virtual COM port layer is added on top of the USB connection and all communications are done through the port. The connection appears as a simple serial connection from the PC's perspective and can be accessed as a TTY device. Communication through a networking protocol such as UDP or TCP has the advantage of being very flexible in terms of the implementation and protocol used. The latency can vary depending on the setup and traffic. The main advantage of this option is the fact that it can be extended for a large amount of different applications and situations.

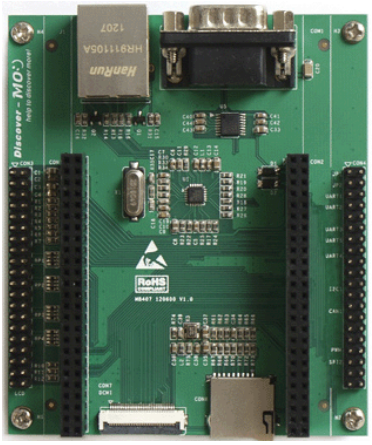

Fig. 5.2 M-STF4BB Base Board

The PC running the Scilab simulation software was is equipped with an Intel Core i5-2400 running at 3.10 GHz with 8 GBs of memory. To execute the control algorithm, the STM32F4 [73] was chosen due to its high clock rate, powerful on-chip peripherals with their associated peripheral drivers and ample documentation. The available software stack for this platform also allowed for quick setup of the test environment. The ICs demonstration board, the STM32 F4-Discovery board, provides access to many external peripherals that are used in the experimentation, such as the USB port. It also allows for the use of the Embest Ethernet expansion board such as the one picture in Figure 5.2. This expansion board allows for a quick deployment of a UDP server.

5.3.3 Software Setup

The conceptual model of the whole system is pictured in Figure 5.3. The simulation contains a reasonably simple plant represented by a continuous-time block running alongside, and interacting with, a discrete-time block that serves as an interface to the microcontroller. The discrete block represents a piece of C code that executes when an input discrete event occurs. The piece of C code contains the callback that performs computations on the set of inputs and produces a set of outputs. In this case, the code contains system calls which sends information about the plant to the hardware and receives the feedback from the controller. This effectively creates a closed-loop feedback system. It is pictured in Figure 5.4.

On the simulation side, it was not necessary to create overly complicated continuous models since most of the objective of the experiment is to verify the capability of the hardware blocks to keep up with the running simulation. The simulated plant is a weighted object on which a force can be applied. Applying a force on the object causes it to move, which is simulated by the three integral blocks, which calculate the accumulation of displacement and speed as a result of the applied force. The position of the object can be described as

$$s(t) = \int \int a(t)\mathrm{d}t + \int v(t)\mathrm{d}t$$

where a is the acceleration, v is the velocity and s is the position at time t, which is what is being computed in the simulation.

The simulation relays the information about the position of the mass to the input block, which sends the position of the object to the hardware on which a control algorithm

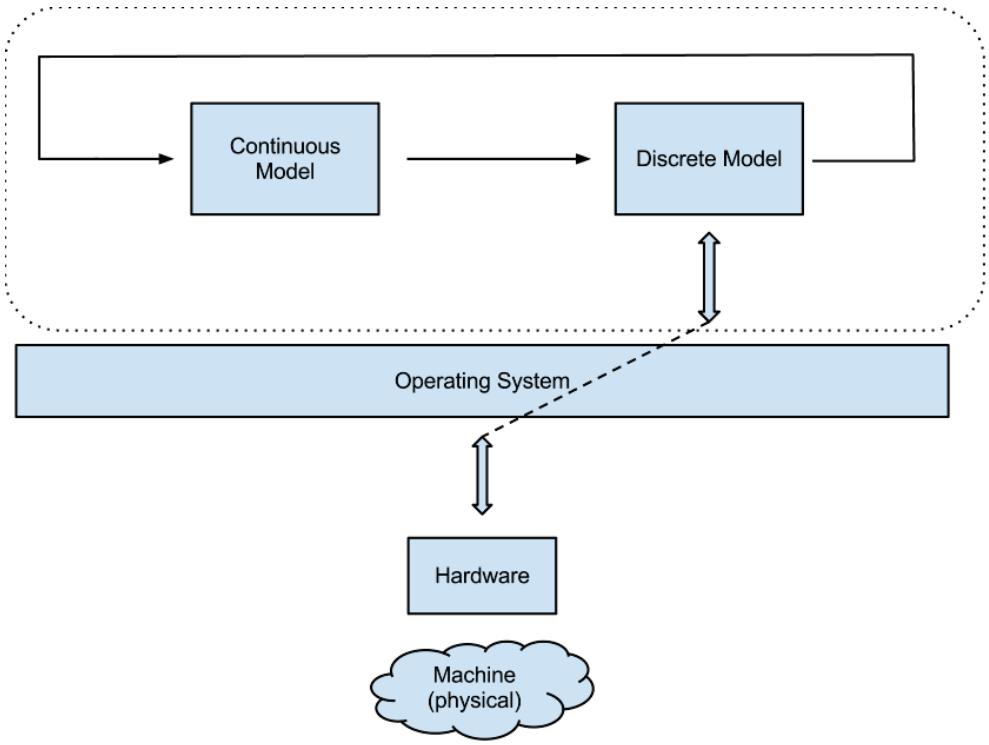

Fig. 5.3 Overall System Configuration

is running. The hardware responds with a desired output force which is applied to the object, effectively closing the feedback loop.

The discrete event generators trigger the callbacks of the discrete function blocks and can be adjusted to have the discrete block executed at a particular frequency. In our particular case, the sampling frequency is set at 1 kHz, which is a commonly found update frequency in industrial control. On the microcontroller's side, things were also kept as simple as possible. The microcontroller would either execute a bang-bang or PID controller. The ChibiOS real-time operating system (RTOS) was chosen in order to have a structure for guaranteed real-time response as well as the software stack for the microcontrollers peripherals. It is one of the best real-time operating systems in terms of code size and real-time performance. The benchmarks for jitter and context switching are very good compared to some other operating systems [74]. To have access to an IP stack on the STM32F4, the minimalist lwIP [75] was used as well as an example project containing an already-made echo UDP server. The echo server was modified to read the contents of packets and return the output of a control algorithm such as PID.

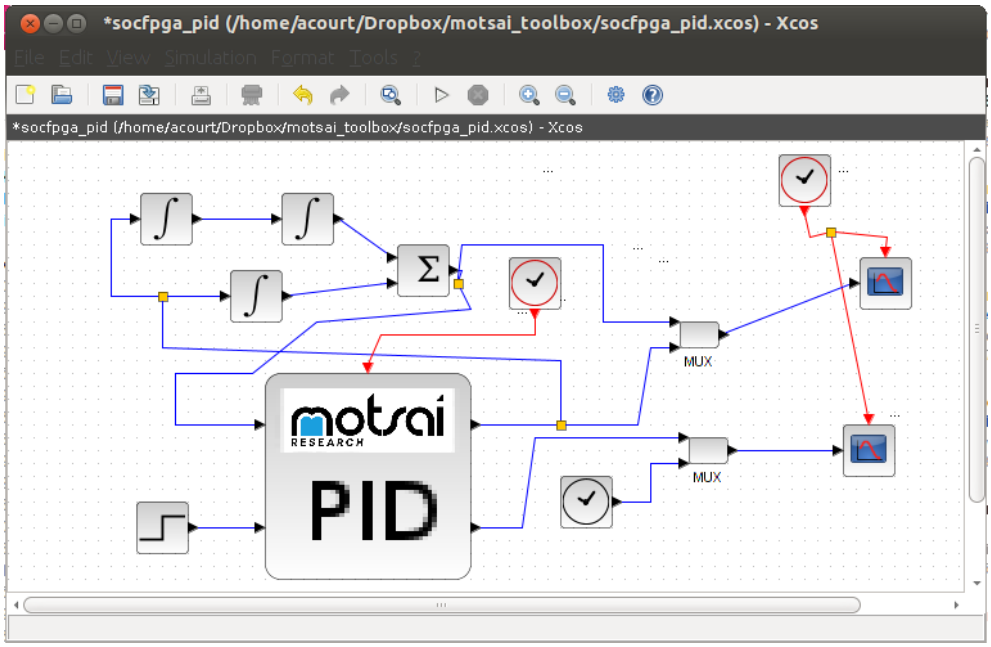

Fig. 5.4 Graphical Xcos Window

Communicating to the external hardware through UDP turned out to be more complicated than through USB. An ethernet card had to be connected to the PCI bus on the Linux PC so that it can access the internet and communicate with the microcontroller at the same time. The OS also had to be configured such that all packets sent to addresses having a certain range would end up on the microcontrollers side. The simulation software does not communicate directly with the microcontroller and send the packets itself. Instead, it communicates with a separate program running concurrently using shared memory. The exchange of information between simulation software and the relay program is arbitrated using *pthreads*. This program then relays this information to the networking stack and expects the response from the microcontroller. Once the response packet from the microcontroller comes back, it forwards it to the simulation. The information flow is pictured in Figure 5.5. This configuration has the advantage of abstracting away the medium of communication between the computer and the STM32F4 from the point of view of the simulation software. As a result, the communication medium used by the relay is interchangeable with others.

As the simulation is executed, there is a divergence that accumulates between the time passing in Scilab and actual physical time. This is a crucial measure for HIL systems

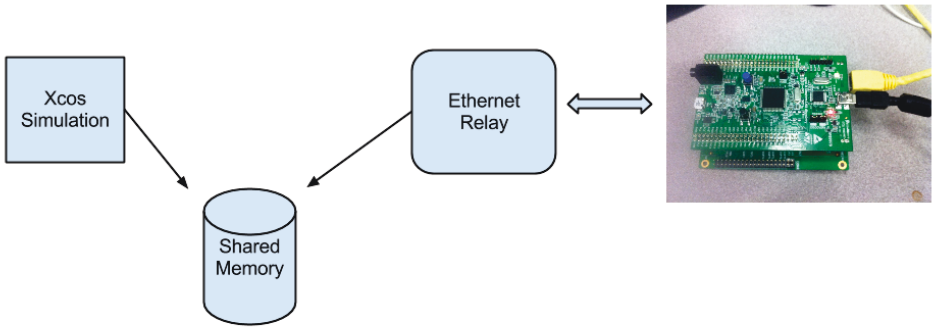

Fig. 5.5 Ethernet Relay

because it can evaluate how closely the simulation of the plant resembles the real thing and how well the algorithm being tested will react. If the discrete time block takes too much time to respond, it will slow down the simulation software to a point where it diverges from reality. The sampling frequency at which the discrete model performs data exchanges with actual hardware has an effect on the physical time and simulation time relationship. If the sampling frequency is too high, the external hardware will not be able to keep up with the simulation software. The discrete model response time is related to the time it takes for an Ethernet packet to leave the PC and reach the microcontroller, get processed by the control algorithm, and get back to the PC. This is called the packet exchange time. The higher the response time, the bigger the time divergence. Furthermore, the jitter of the response time from the hardware can cause some problems for the performance of the real-time system. The interested reader can read a publication by Yu et al. that attempt to analyze the effect of jitter on the performance of PI and PID controllers [76].

To summarize, this experiment examines the following metrics:

1. Simulation time versus actual time
2. Maximum Sampling Frequency
3. Packet exchange time
4. Jitter of the response time

5.3.4 Experimental Results and Analysis

USB Virtual COM Port

The speed performance of the exchange of information between the simulation and the hardware when using USB communication turned out to be disappointing. In fact, the maximum functional sampling rate that could be achieved was 5 Hz before the simulation time and physical time started diverging. 5.3.4 shows how at 5 Hz, there is an increasing differential between simulation time started diverging from the beginning. The staircase shape of the real-time curve is simply the result of the simulation blocking before updating this metric on the scope and the relatively slow update frequency of the running simulation. In other words, the simulation software calculated an elapsed simulation time of 5 seconds, but by querying the host computer, it can show that the actual time that has elapsed is actually 7 seconds.

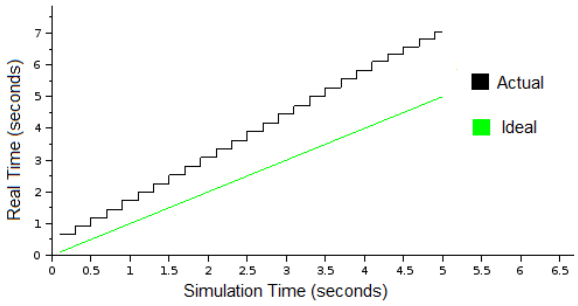

Fig. 5.6 Real Time vs. Simulation Time Plot for USB at Fs = 5 Hz

The slow response time can be partly explained by the fact that the USB Virtual COM driver will usually wait for multiple characters before sending out a string in the stream buffer. This is problematic if you only want to send a few characters each time (as is the case in our situation). The most optimal solution would be to create an HID driver from scratch. However, even with a barebone USB solution, it won't perform much better due to USB buffering. In any case, USB has the disadvantage of being unreliable over long distances and less flexible. As a consequence, the possibility of using USB has been abandoned.

Ethernet/UDP

It turns out that the performance of UDP is actually much better than the performance of USB. When running at 1 kHz, which is pictured in Figure 5.8, the real time and simulation times follow each other much more closely. In fact, there is reasonable time correlation when setting the sampling frequency up to 3 kHz. However, the time metrics start to dissociate when setting the frequency to more than 5 kHz (Figure 5.9). A kilohertz sampling frequency is viable for simple control systems.

Fig. 5.7 Real-Time vs. Simulation Time for UDP

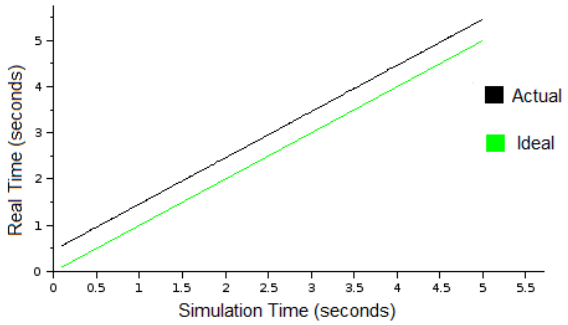

Fig. 5.8 Fs = 1 KHz

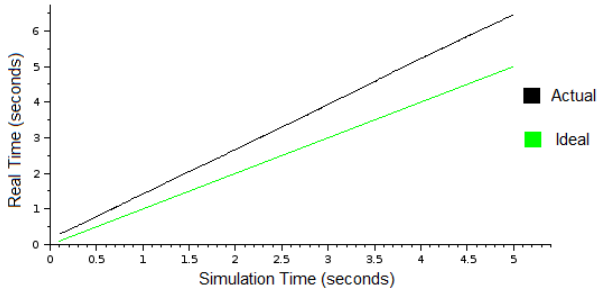

Fig. 5.9 Fs = 5 KHz

Response time jitter for UDP packets

Most implementations of feedback control systems can be susceptible to timing jitters. Thus, an important metric to be analyzed is the jitter of the response time. A varying sampling frequency can affect the operation of the control algorithm. Generally, the effect of jitter is to produce a frequency-selective attenuation as well as a uniform spectral density

component in the resulting signal [77]. This can be detrimental to the operation of the control algorithm.

The objective of the experiment is to establish how much variability in the response time there is in this specific HIL situation when using UDP as the means of communication between the simulated plant and the real-time controller. The main causes of jitter in this situation are the context of the execution of the kernel, the network hardware and the network traffic. Since the testbench comprises of a system running on a non-deterministic OS, it is entirely possible that control packets are delayed due to unrelated activites of the computer system (for example, a user saving a file during the running simulation). There are ways to reduce the effects of these events such as optimizing the algorithm of the relay or by increasing the priority of the simulation process.

To monitor the packet exchange time, all the packets are given a specific tracking number and Wireshark [78] is used to track the leaving and returning of individual packets on the network. The analysis of the network exchange lasts 5 seconds of simulation time. Figure 5.10 is a histogram of the times it took for a packet to be sent to the UDP server and received back. The majority of the packet exchange times lie around the 0.1 millisecond mark. However, there are many outliers in the range between 800 microseconds to 1 millisecond, and a bit more in the 1 millisecond to 1.5 millisecond range. This is problematic because a minimum stable sampling frequency of 1 KHz is needed for any serious control application.

To remedy this, optimizations on the microcontroller side can provide major improvements to the consistency of the distribution of the deltas. An improvement to the Ethernet relay algorithm to make it consume less CPU cycles also had an effect. The improved results can be seen in Figure 5.11:

With the improvements, most packets end up having a delta in the range of 50 to 150 microseconds. The outliers only reside in the range of 200 to 216 microseconds (the maximum being 216 microseconds). This shows that the network and STM32F4 parts of the feedback loop occupy a small portion of the feedback loop. Therefore, a sampling frequency of 1 KHz can easily be sustained if it is needed.

Comparison with other HIL testbenches

This HIL test bench is similar to what has been done in other research projects. For example, Abugchem et al. present a real-time HIL simulation platform [79]. Similar to our

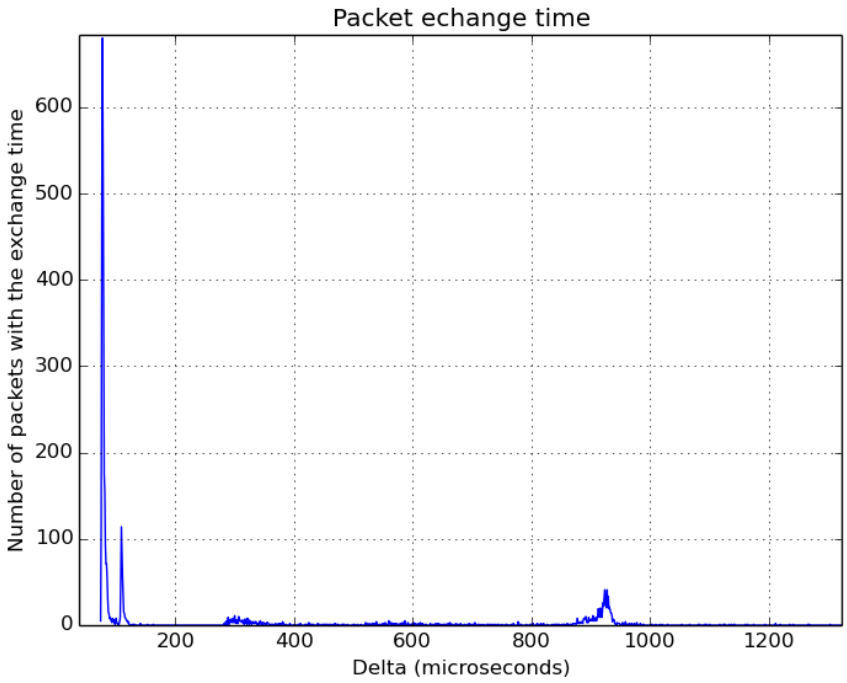

Fig. 5.10 Packet Exchange Time Histogram at Fs = 1 KHz

platform, it uses a simulated model build on a desktop PC that is connected to the embedded real-time processor. The outputs of the PC are produced according to a simulated model and sent to a controller and controller behaviour is sent back to the PC to close the control loop. However, the focus of their experiment is different. They want to use this testbench to test the effects of different scheduler schemes on the behaviour of the controller. The direction of their research has continued on this path with more publications that explore different configurations such as the parametrization configuration of the control algorithm [55]. By contrast, this experiment is designed to test the limits of this testbench setup. They examine the response jitter caused by different scheduling schemes and computing loads on the embedded controller but they have not taken into account the possibility of errors due to the jitter in the communication link between the PC and the embedded controller. Another important difference is the set of software that is used. In their test bench, which is based on the RTE-SIM environmment [80], they mostly use proprietary software which is less flexible than open-source. They also base their experimental setup on a Windows machine. By contrast, in this test bench, all of the software

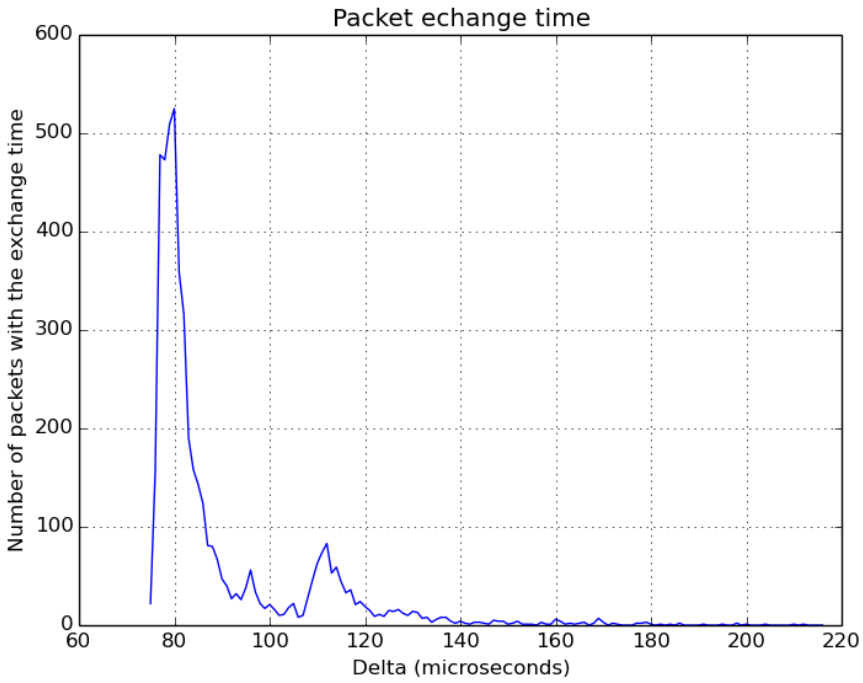

Fig. 5.11 Packet Exchange Time Histogram with improvements at Fs = 5 KHz

tools used on the desktop are open-source. By leverage an open-source OS (Linux) and simulation software (Scilab/Xcos), it was possible to have more control and to maximize the performance of the test bench setup.

5.4 Using Scilab and Xcos on the SoCFPGA

Simulation software is a valuable addition to a development environment. In the case of the *Golden Lion* platform, including simulation software as part of the development suite can introduce interesting additions to the workflow. In this section, it will be shown how running Scilab and Xcos on the SoC FPGA processor can open up possibilities for a design workflow that does not require coding skills. This section will show how a control system engineer will be able to develop an algorithm using the graphical features of Scilab/Xcos and deploy it to the rpoposed platform easily. Additionally, this section will examine the possibility of using Scilab for an as/more reliable HIL test bench than the one based on a desktop PC and an external microcontroller presented in the previous section.

5.4.1 Workflow proposal

To help designers build efficient applications for the *Golden Lion* platform using Scilab, a Scilab and Xcos toolbox is built to give access to computation accelerators on the FPGA through simple function calls. This MotSAI Scilab toolbox is usable both on the embedded platform and on the desktop. The toolbox would contain a set of API calls corresponding to what is contained in the FPGA. A workflow based around this Scilab toolbox is proposed. First, consider a use case in which an engineer wants to design a control algorithm which takes in input signals, performs computations on these signals and produces the appropriate output control signals. An example of this could be a developer who would like to create an output waveform with a set of frequency components. In this scenario, the developer develops the initial design on a desktop computer running Scilab. In the Scilab program containing the application, calls are made to the API to activate the waveform synthesizer features on the FPGA. The next step would be to transfer the *.xcos* or *.sci* file to the SoC FPGA and open it in Scilab from there. Using graphical *.xcos* files assumes that there the SoC FPGA is connected to a keyboard and mouse for user input as well as a connected screen. Once opened in Scilab, the simulation appears as though it were designed on the desktop, except that the hardware will behave as intended. The developer executes the program and observes the behavior of the HIL and the plant simulation. Depending on the behaviour of the program (if there is a bug or uninteneded behaviour), the developer would perform changes on the desktop and restart the iteration loop.

It should be noted that Xcos is not essential to this workflow since it is essentially a graphical wrapper around Scilab. One could easily design their algorithm entirely in Scilab and deploy it on the SoC FPGA as is. If it is decided that Xcos it not to be used, it entirely removes a layer of complexity on the runtime of the application and will improve execution speed.

5.4.2 Hardware-in-the-Loop Simulation with SoCFPGA

In the previous section, an experimental testbench was set up between an external PC and a microcontroller to determine the reliability of a HIL setup for testing an algorithm running on a micro-controller. This section will explore this concept further, except that the simulation and the external hardware will be further integrated by running both on the same chip. The SoC FPGA chip that is used in this setup has powerful features, including

a ARM Cortex-A9 dual-core processor that is capable of running Ubuntu as well as a large FPGA fabric. Having such a processor allows it to run simulation software such as Scilab at a reasonable speed. However, the main advantage of transferring the testbench to the SoC FPGA is the fact that the information transfer between the unit being tested and the running simulation will undergo less jitter and will perform at a lower latency. The fact that the unit tested is in the FPGA and the simulation runs on the processor means that information travels through memory transfer rather than through an external bus such as Ethernet or USB. The downside of this HIL setup is that the simulation software does not run as fast on the ARM Cortex-A9 as it does running on a Intel Core i5™. This approach may simply move the system bottleneck from the information transfer bus to the processor. This subsection will examine whether putting the HIL test setup on the SoC FPGA improves the performance of the testbench or makes it worse.

A new simulation file is used in this particular setup. This Xcos simulation is mostly the same as in the previous section. The simulated plant and the hooks to plot the real vs. simulated times are the same. Keeping track of real time is done through a system call every time an execution of the discrete time block is made. The test runs for 25 seconds. However, instead of making the PID block be a communication to an external processor, it is a connection to a PID algorithm implemented on the FPGA fabric. The metric used to determine the viability of a HIL platform is whether the simulation time diverges from the real time. To do this, the simulation is run at different frequencies to determine the exact sampling rate at which it breaks down.

As it turns out, having the HIL set up on the ARM processor core does not allow for a greater sampling frequency than having the HIL setup on the desktop. The transfer of information between the simulation and the hardware is almost instantaneous (a single memory instruction) and the computation of the PID algorithm takes 3 clock cycles at 50 MHz the execution of the simulation software is considerably slowed down by the heavy load of the simulation software (the graphical application). Figure 5.12 shows the divergence between simulation time and real time when executed at different sampling frequencies. The simulation runs fine at 100 Hz sampling frequency. However, as the sampling frequency is increased, the divergence starts appearing, as can be seen in Figure 5.13 to Figure 5.14. A possible solution to this problem is to segregate the graphical and real-time portions of the simulation software to the two different cores.

The conclusion of this experiment is that this form of HIL testbench involving only a

Fig. 5.12 Real-Time vs. Simulation Time on ARM Cortex-A9

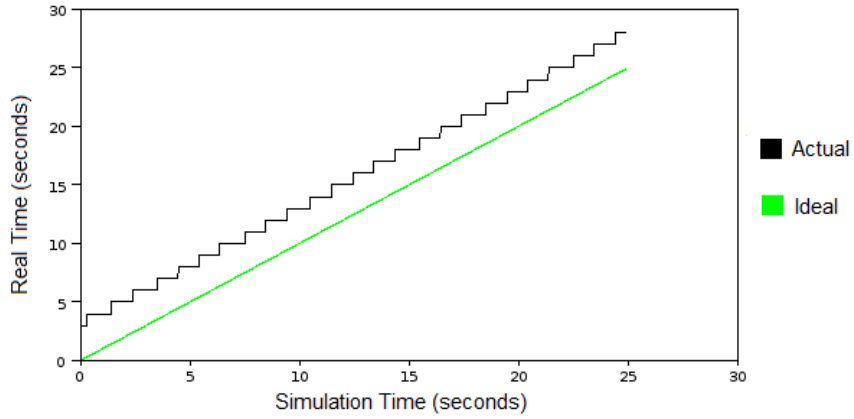

Fig. 5.13 Fs = 100 Hz

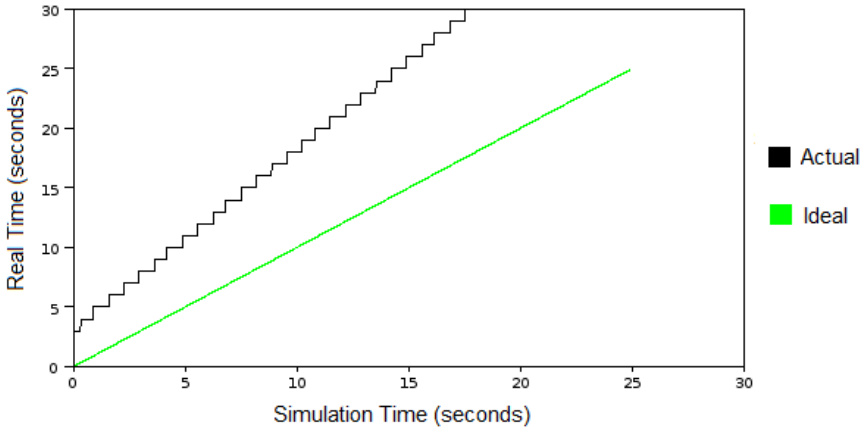

Fig. 5.14 Fs = 166 Hz

SoC FPGA is not as effective as on a desktop computer with an external connection to the unit under test, despite the downsides of having a relatively slower link between the two in the latter case. The main problem is the lack of speed of the ARM Cortex-A9 present on the SoCFPGA. Although the HIL and the link to it are very fast, the simulation software as well as its graphical portion Xcos is too much of a burden on the processor that is mainly intended for embedded applications. However, it is worth noting that in the near future, SoC FPGAs might get a serious upgrade due to the recent partnership with Intel to fabricate their chips [81]. Although the deal is limited to manufacturing technology for now, if Altera decides to incorporate x86 processors inside their FPGAs or if Intel uses programmable logic in their CPU (the area of the die would probably be dominated by the x86 core), it would boost the processing power, allowing for graphical simulation applications like Xcos to run more smoothly.

5.5 Register Map Viewer

In addition to the collection of tools allowing for development with HIL simulation, there is a need for tools meant for debugging device drivers interacting with peripherals on the SoC. This section presents the development of a tool for interacting and debugging peripherals that are described by a SVD file and that are accessible by a CPU's memory map.

5.5.1 Motivation

Because of complex integration of multiple components, SoCs contain many peripherals that perform different functions. In order for these peripherals to be used by the system developer, the OS has to take them in charge by the means of a device driver, which are computer programs that control peripherals that are attached to the computer bus. However, sometimes the device driver does not function correctly and contain bugs, so it can become necessary for the developer to bypass the device driver and understand what is going on methodically.

To understand the behaviour of a memory-mapped peripheral and debug it, one must read and write to its registers. The manual debugging method is not trivial and can open the way for many mistakes to be made. For example, if there is unwanted behaviour from a peripheral and the developer wants understand and fix the problem.

The workflow for this situation would be the following:

1. Look up the address of a particular register from the datasheet in the peripheral that needs to be debugged.
2. Find the bit offset of a field you want to read from.
3. Write a program that reads and writes to the system physical memory.
4. Use this program to read and write to specific fields in the registers of the peripheral and translate the results in hexadecimal human input.
5. Interpret the hexadecimal output of the program and understand the bug.

Every step in this workflow is vulnerable to small errors. The biggest problem is that it involves humans interpreting hexadecimal values and inferring the register content meaning from those values. This process is very prone to human error and those errors can have dire consequences and be very difficult to locate as they pertain to device drivers.

5.5.2 Features

To address this complexity, a tool called the Register Map Viewer has been created to assist the developer with correctly viewing the register map defined by a chip manufacturer as well as help the developer read and write to specific fields in a register.

In order to obtain information about the register map of the device, the tool parses a CMSIS-SVD file [44] that is given by the manufacturer or that is automatically generated based on the synthesized contents of an FPGA. The SVD parser is derived by one built by Ben Nahill [82]. More functions were added to take into account the possible values a register could take with the use of enumerated values. The auto-complete function automatically shows the possible options for the different peripherals, registers, fields, and values according to what the user chooses. This way, the developer will avoid having to look through documentation to figure out the address and bit offset of the field he wants to write to, thus reducing the risk of errors. Using this tool, it becomes much simpler to read and write to specific registers.

Figure 5.15 provides an example use case in which a user would want to write a character to the transmit buffer of a UART peripheral present on a Freescale i.MX6Q processor. With this tool, it becomes much easier to perform this task, since it already contains much of the information contained in the register documentation. First, the user loads a preparsed

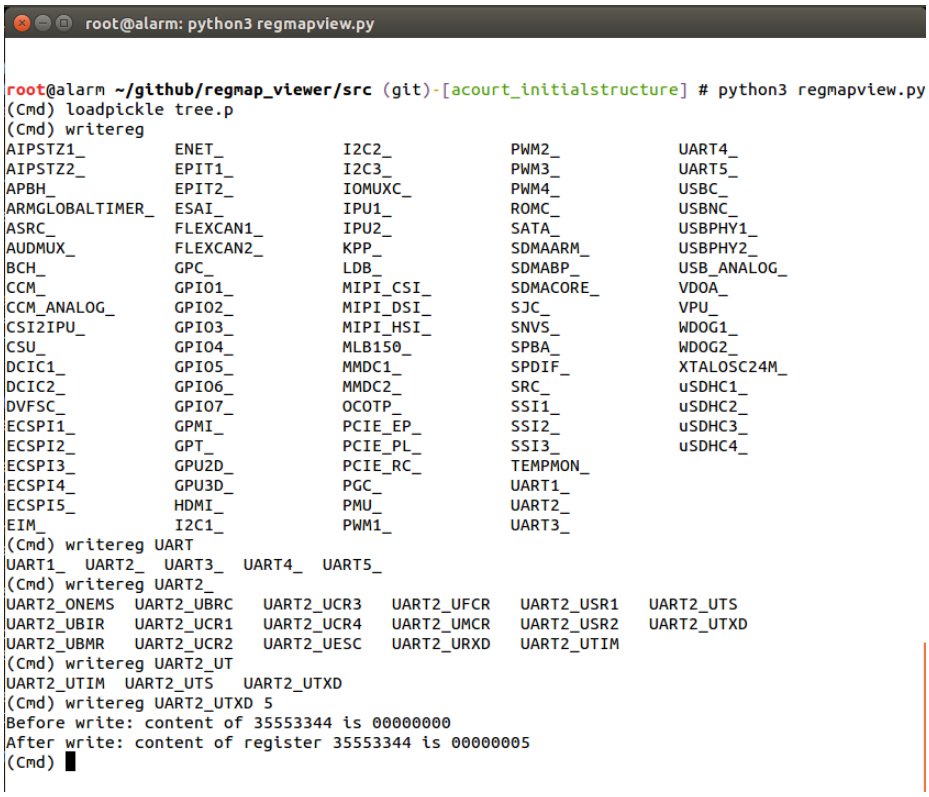

Fig. 5.15 Writing to a memory-mapped register

python object tree containing the information about the register map into the memory. Then, the user types the command to write to a register (*writereg*) and presses *tab*. This makes the program show all of the available peripheral prefixes. The user continues to refine his choice using the auto-complete features until the full name of the intended peripheral is chosen. The program then offers the user a choice of different registers contained in that peripheral, and the auto-complete helps the user type out the complete intended register name. The user then types the desired value and executes the write command.

Another important feature of this tool is reading a memory-mapped register. An example use case of this would be when a user wants to read the contents of a UART control register, which is pictured in Figure 5.16. The initial program flow is the same as in the case in which the user would need to write to a register, and functions the same way in terms of helping the user find the intended register name. After the execution of the read command, the contents of the chosen registers are displayed. The hexadecimal value is interpreted according to the SVD file and each field is explained, eliminating the need for

constantly reviewing the documentation.

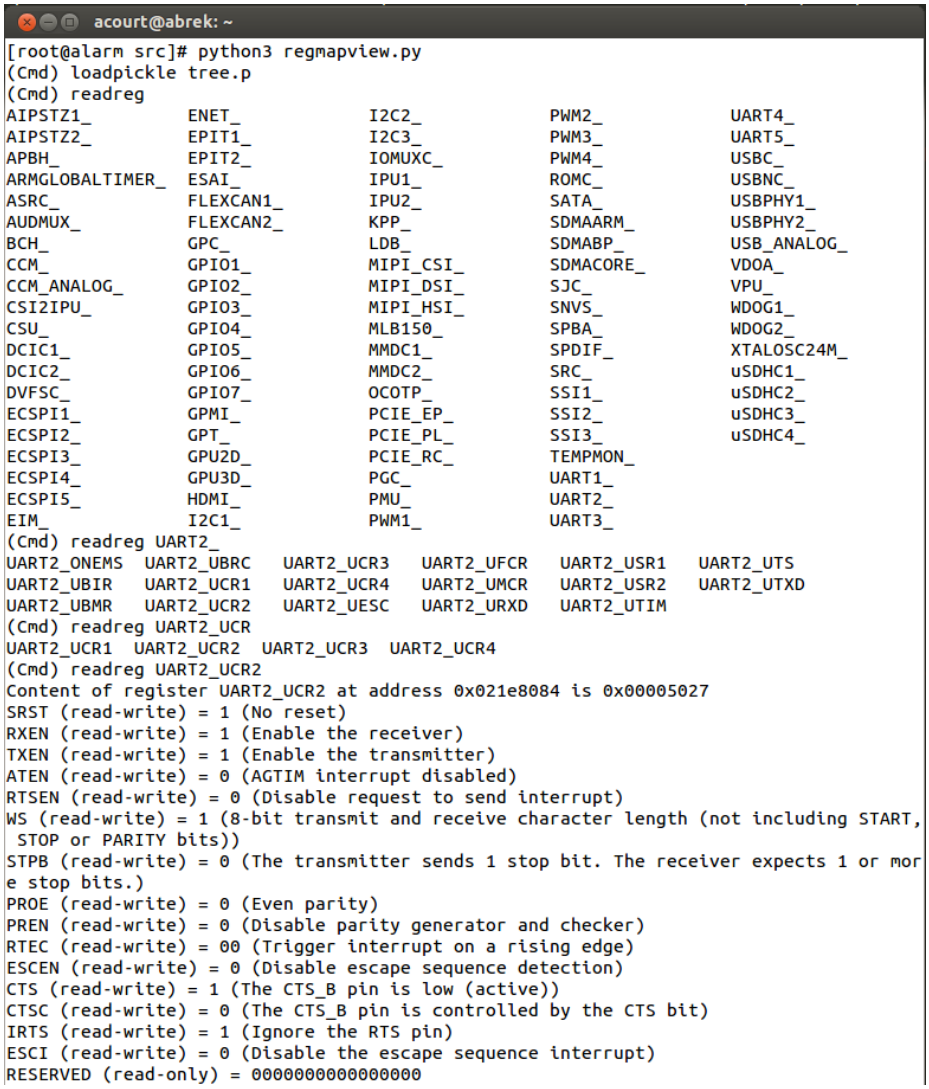

Fig. 5.16 Reading a memory-mapped register

5.5.3 Design

The tool is accessible by the user as a Python script [45]. The parsing of the of the SVD file is done using the lxml library [83] which automatically generates a python object file containing the information about the tree structure of the XML. However, parsing an xml file can take a big amount of processing power depending on the size of the file. Since SVD

files are generally very big (the one for the i.MX6Q is 15.5 MBs, which is huge for a text file), parsing them can take a long time. In order to avoid performing such a workload on an embedded system, the parsing of the tree can be done externally (generally on a high-powered desktop or server) and deployed on the embedded system in the form of a *pickle object*. Loading a python pickle object takes a fraction of the time as it does loading and parsing the original XML file.

The part of the program that accesses the physical memory is written in C and is compiled into a dynamic library. This section is decoupled from the parts that handle the user interface and the management of the register database. Since programs are forbidden from accessing areas outside of the allocated virtual memory space, access to the physical memory is done by performing a memory-map *mmap*() call to */dev/mem*. The python script portion of the program accesses the dynamic library using the *ctypes* module. In order to deploy a command-line interpretation feature, the python *cmd* module is used. This module provides a framework for creating command-line interpreters and provides features such as auto-complete, command memory, and many other tools that are regularly used in command-line applications.

5.5.4 Comparison with other tools

Looking at registers of a running system is nothing new. There are many modern commercial debugging utilities that are more sophisticated than the one presented [84] [85]. However, they are expensive and require non-negligible amount of processing power to use. The Register Map Viewer is one of the only command-line tools that is lightweight enough to run *on the target system* and that can be used to view the content of registers at runtime. By comparison, some other tools have been made run on a remote PC and transmit register information via *gdb* [82]. By running on the target system, the program allows for much greater flexibility. For example, it allows the user to debug running applications without necessarily having access to a PC or a JTAG debugger. This means that a developer has access to features of a modern debugging tool and can fix problems remotely with the use of a terminal.

5.5.5 Possible improvements

Although being able to run through the terminal is one of the key features of this tool, the core of the program is decoupled enough from the user interface features that it would be possible to use the core debugging features with a GUI. Instead of showing the different registers and fields in a terminal, they could be shown in a window and changed using buttons and text fields similar to how it is done in a modern commercial IDE like Keil uVision [85] and ARM DS-5 [84].

It would also be interesting to explore the possibility of leveraging the capabilities of an FPGA to store and transmit massive amounts of data for the purpose of debugging. Being able to monitor the contents of an FPGA register over an amount of time and being able to access this information from a program running on the embedded processor could be useful in some situations. The functionality would be similar to how the internal logic analyzers like Altera's SignalTap [59] and Xilinx's ChipScope [86] work, but it would be gathering data and sent to a userspace program instead of to a PC connected by JTAG. As an example use case, a developer could want to monitor the behavior of an FPGA peripheral that is being controlled by a Linux kernel module. An FPGA would be programmed so that it would hold trace data relevant to a certain peripheral (a sort of "debug" compilation mode) in a buffer for a fixed number of clock cycles. The program, running in userspace, would then take in the samples and visually show them to the user to help the debugging process. Although this would require recompiling the bitstream in a "debug" mode and can take a lot of time, FPGA reconfiguration techniques exist that allow the debugging loop to be shortened [87].

Chapter 6

Conclusion

6.1 Summary

In this master's project, the basis for Motsai Research's next generation computing platform for industrial embedded systems has been proposed and prototyped. By leveraging a computer system based on SoC FPGA technology, it was possible for the system to be at the cutting edge of development in the electronics industry. The solution's architecture is based on a CPU, an FPGA, and a soft real-time microcontroller. It is sophisticated enough to be able to fulfill both low-level and high-level computing roles. The design of basic IP cores such an NCO, a PID and LMS based adaptive filter has been described. The main academic contribution of this thesis is the demonstration of open-source simulation software performing HIL simulation on a SoC FPGA. The possibility of using free software for HIL simulation allows engineers to build embedded systems more quickly and expand the platform with adapted components.

6.2 Future Work

Although its basis is functional and usable, the *Golden Lion* platform constitutes a flexible baseline on which to build an end application or sophisticated modeling tool. Improvements can be made on multiple fronts. The most obvious improvement is to use the research in this thesis to make a product destined for industrial applications. It is important to build a basic module containing the SoC FPGA and at minimum one bank of DDR3 memory. This module can be connected to a I/O PCB containing the electrical protections and isolation

appropriate for industrial applications. For the real-time microcontroller being synthesized inside the logic fabric, its Arithmetic Logic Unit (ALU) can be optimized to use less resources when executing fixed-point operations using transforms for imprecise datapaths [88] [89]. This could be an interesting avenue for future work. Interested readers may want to learn about methods for verification of datapath circuits through error modeling [90].

Improvements can also be made on the software side. First, the existing IP cores can be improved so that they have more features and have more comprehensive software drivers. Many other peripherals and drivers can be created. Next, because of the inevitable future improvements in SoC FPGA technology, the HIL testbench running in Xcos and Scilab should be able to run more smoothly than what has been shown in the test run in section 5.4.2. The register viewer tool presented in Section 5.5 can be improved by creating a graphical interface around it, or by giving it the ability to handle larger amounts of register information.

Appendix A

F4-Discovery Daughter Board Schematics

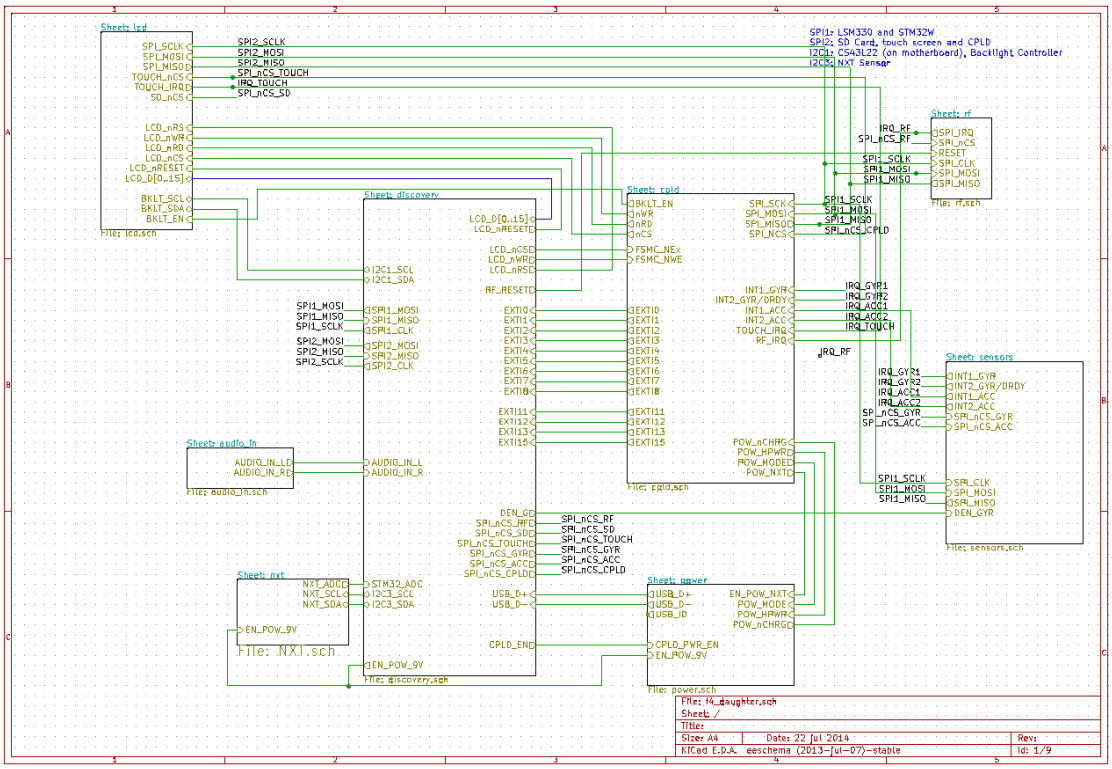

Fig. A.1 STM32F4-Discovery Board

Appendix B

Python Code

B.1 LUT Memory Content generation

```
import math

f = open('sinewave.txt','w')
data_width = 12;
addr_width = 12;
rom_length = 2**addr_width;
scaling_factor = (2**(data_width-1))-1
sine_wave = [ scaling_factor * \
(1+math.sin(2 * i * math.pi/rom_length)) for i in range(rom_length)]

for x in sine_wave:
        f.write("{0:012b}\n".format(int(x)))http://only-vlsi.blogspot.ca/200

f.close()
```

B.2 PID python script

```
from devmem import *

bridge_base_addr = 0xff200000
base_offset = 0x0000
pid_offset = bridge_base_addr + base_offset

class pid(object):
        """docstring for interacting with the PID"""

        registers = {'kp':pid_offset, 'ki':pid_offset+1, 'kd':pid_offset+2,
                        'pid_in':pid_offset+3, 'pid_ref':pid_offset+4,
                        'eval':pid_offset+5, 'pid_out':pid_offset+15}
        def __init__(self):
                self.dm = DevMem()
                self.offsets = [0x0,0x1,0x2,0x3]
                self.resetvalues = [0,1,2,3,4]
                reg = self.dm.read(pid_offset + self.offsets[2])
                if reg == self.resetvalues[2]:
                        print("hurray it works!")

        def verify_pid(self):
                f = open("InY.txt", 'r')
                g = open("InRef.txt", 'r')
                o = open("pid_out.txt", 'r')
                v = open("pid_hardware_out.txt", 'w')
                InY = []
                InRef = []
                pidout = []

                for line in f:
                        InY.append(int(line))
                for line in g:
                        InRef.append(int(line))
```

```
                for line in o:
                        pidout.append(int(line))
                self.start_evaluating()

                i=0
                out=123
                valid = 1
                while(pidout[i] != -1 and i < 20000 and valid == 1) :
                        self.dm.write(self.registers['pid_in'], InY[i])
                        self.dm.write(self.registers['pid_ref'], InRef[i])
                        out = self.dm.read(self.registers['pid_out'])
                        if (i != 0 ) and out != pidout[i]:
                                valid = 0
                        else:
                                v.write(str(out)+'\n')
                        i = i+1
                v.close()

        def set_pid_params(self):
                self.dm.write(self.registers['kp'], 65536)
                self.dm.write(self.registers['ki'], 21475)
                self.dm.write(self.registers['kd'], 6554)

        def start_evaluating(self):
                self.dm.write(self.registers['eval'], 1)

if __name__ == '__main__':
        mypid = pid()
        mypid.set_pid_params()
        mypid.verify_pid()

        print("finished")
```

References

[1] T. J. Talley, "IEEE industrial electronics conference 2003: development process for replacement of an obsolete microcontroller," in *Industrial Electronics Society, 2003. IECON'03. The 29th Annual Conference of the IEEE*, vol. 2. IEEE, 2003, pp. 1552–1556. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1280288

[2] M. Paska, P. Dvorak, S. Racek, and E. Janecek, "Model based support for life cycle management of i&c systems," in *EUROCON, 2007.* IEEE, 2007, pp. 2217–2220. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4400452

[3] X. Meng, B. Thornberg, and L. Olsson, "Component obsolescence management model for long life cycle embedded system," in *AUTOTESTCON, 2012 IEEE.* IEEE, 2012, pp. 19–24. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6334547

[4] J. Torresen and T. A. Lovland, "Parts obsolescence challenges for the electronics industry," in *Design and Diagnostics of Electronic Circuits and Systems, 2007. DDECS'07.* IEEE, 2007, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4295267

[5] W. Ahmed and D. Myers, "Maintainable embedded system design to accommodate incremental change," in *Integrated Circuits, 2007. ISIC'07. International Symposium on.*, 2007, pp. 158–161. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4441821

[6] D. Hallmans, T. Nolte, and S. Larsson, "Industrial requirements on evolution of an embedded system architecture." IEEE, Jul. 2013, pp. 668–673. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6605869

[7] D. Hallmans, T. Nolte, and Larsson, "A method for handling evolvability in a complex embedded system," in *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on.* IEEE, 2013, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6648016

[8] PCI-SIG. (2011) Conventional PCI. [Online]. Available: http://www.pcisig.com/specifications/conventional/

[9] "Parallel vs serial data transmission," 2008. [Online]. Available: http://only-vlsi.blogspot.ca/2008/04/parallel-vs-serial-data-transmssion.html

[10] PCI-SIG, "PCI express 3.0 frequently asked questions," 2012. [Online]. Available: http://www.pcisig.com/news_room/faqs/pcie3.0_faq/PCI-SIG_PCIe_3_0_FAQ_Final_07102012.pdf

[11] L. Benini and G. De Micheli, "Networks on chips: a new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=976921

[12] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Proc. DAC*. IEEE, 2001, pp. 684–689. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=935594

[13] ARM, "Cortex-a series," Jul. 2014. [Online]. Available: http://www.arm.com/products/processors/cortex-a/

[14] Altera, "Cyclone v device overview," Dec. 2013. [Online]. Available: http://www.altera.com/literature/hb/cyclone-v/cv_51001.pdf

[15] J. Beneke and Avnet, "Designing with the xilinx 7 series PCIe embedded block," Apr. 2012. [Online]. Available: http://www.em.avnet.com/en-us/design/trainingandevents/Documents/X-FEST%202012%20PRESENTATIONS/xfest12_pdf_pcie_v1_1_april29.pdf

[16] Y. Cao, Y. Zhu, X. Wang, J. Jiang, and M. Qiu, "An FPGA based PCI-e root complex architecture for standalone SOPCs." IEEE, Apr. 2013, pp. 149–152. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6546010

[17] W. H. Wolf, "Hardware-software co-design of embedded systems [and prolog]," *Proceedings of the IEEE*, vol. 82, no. 7, pp. 967–989, 1994. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=293155

[18] G. De Michell and R. K. Gupta, "Hardware/software co-design," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 349–365, 1997. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=558708

[19] Y. Zou, Z. Zhuang, and H. Chen, "HW-SW partitioning based on genetic algorithm," in *Evolutionary Computation, 2004. CEC2004. Congress on*, vol. 1. IEEE, 2004, pp. 628–633. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1330916

[20] H. P. Peixoto and M. F. Jacome, "Algorithm and architecture-level design space exploration using hierarchical data flows," in *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*. IEEE, 1997, pp. 272–282. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=606833

[21] Y. Chu, "Application-specific coprocessor computer architecture," in *Application Specific Array Processors, 1990. Proceedings of the International Conference on*. IEEE, 1990, pp. 653–664. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=145500

[22] P. W. Diodato, J. A. Fields, M. E. Thierbach, and M.-S. Tsay, "The design of an IEEE standard math accelerator unit," *Solid-State Circuits, IEEE Journal of*, vol. 20, no. 5, pp. 993–997, 1985. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1052426

[23] Y. Chu and K. Itano, "A top-down parsing co-processor for compilation," in *System Sciences, 1989. Vol. I: Architecture Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, vol. 1. IEEE, 1989, pp. 403–413. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=47182

[24] O. Queinnec, "A graphics co-processor and its display processor ICs," *Consumer Electronics, IEEE Transactions on*, no. 4, pp. 551–556, 1987. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4071595

[25] J. H. Schiller, "A flexible co-processor for high-performance communication support," in *Global Telecommunications Conference, 1995. GLOBECOM'95., IEEE*, vol. 2. IEEE, 1995, pp. 1445–1449. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=502641

[26] W. W. Loh and F. J. Dickin, "A novel computer architecture for real-time solution of inverse problems [electric impedance tomography]," in *Advances in Electrical Tomography (Digest No: 1196/143), IEE Colloquium on*. IET, 1996, pp. 22–1. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=578016

[27] D. Niehaus, K. Ramamritham, J. A. Stankovic, G. Wallace, C. Weems, N. Burleson, and J. Ko, "The spring scheduling co-processor: Design, use, and performance," in *Real-Time Systems Symposium, 1993., Proceedings.* IEEE, 1993, pp. 106–111. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=393510

[28] J. Starner, J. Adomat, J. Furunas, and L. Lindh, "Real-time scheduling co-processor in hardware for single and multiprocessor systems," in *EUROMICRO 96. Beyond 2000: Hardware and Software Design Strategies., Proceedings of the*

22nd EUROMICRO Conference. IEEE, 1996, pp. 509–512. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=546476

[29] M. Vetromille, L. Ost, C. A. Marcon, C. Reif, and F. Hessel, "Rtos scheduler implementation in hardware and software for real time applications," in *Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on*. IEEE, 2006, pp. 163–168. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1630765

[30] R. Hartenstein, J. Becker, and R. Kress, "An embedded accelerator for real-time image processing," in *Real-Time Systems, 1996., Proceedings of the Eighth Euromicro Workshop on*. IEEE, 1996, pp. 83–88. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=557821

[31] O. A. Nava and A. D. Prez, "Acceleration of fractal image compression using the hardware-software co-design methodology." IEEE, Dec. 2009, pp. 167–171. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5382046

[32] X. Niu, L. Galarza, Y. Gao, and J. Fan, "Software-hardware co-design for video coding acceleration," in *System Theory (SSST), 2012 44th Southeastern Symposium on*. IEEE, 2012, pp. 57–60. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6195122

[33] Massicotte, D Barwicz, A., "An application-specific processor dedicated to kalman-filter-based correction of spectrometric data," vol. 1, Hamamatsu, May 1994, pp. 352 – 356.

[34] D. E. Borth, I. A. Gerson, J. R. Haug, and C. D. Thompson, "A flexible adaptive FIR filter VLSI IC," *Selected Areas in Communications, IEEE Journal on*, vol. 6, no. 3, pp. 494–503, 1988. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1917

[35] O. Gay-Bellile and E. Dujardin, "Architecture of a programmable FIR filter co-processor," in *Circuits and Systems, 1998. ISCAS'98. Proceedings of the 1998 IEEE International Symposium on*, vol. 5. IEEE, 1998, pp. 433–436. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=694525

[36] M. Berekovic, P. Pirsch, T. Selinger, K.-I. Wels, C. Miro, A. Lafage, C. Heer, and G. Ghigo, "Architecture of an image rendering co-processor for MPEG-4 systems," in *Application-Specific Systems, Architectures, and Processors, 2000. Proceedings. IEEE International Conference on*. IEEE, 2000, pp. 15–24. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=862374

[37] IBM, "IBM systems cryptographic hardware products," Apr. 2014. [Online]. Available: http://www-03.ibm.com/security/cryptocards/

[38] R. McMillan, "Microsoft supercharges bing search with programmable chips," Jun. 2014. [Online]. Available: http://www.wired.com/2014/06/microsoft-fpga/

[39] Intel, "Intel 64 and IA-32 architectures software developers manual," Sep. 2013. [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html?iid=tech_vt_tech+64-32_manuals

[40] A. Hodjat and I. Verbauwhede, "Interfacing a high speed crypto accelerator to an embedded CPU," in *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, vol. 1. IEEE, 2004, pp. 488–492. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1399180

[41] R. A. Melo, D. M. Caruso, and S. E. Tropea, "Memory-mapped i/o over dual port BRAM on FPGA," in *Programmable Logic (SPL), 2012 VIII Southern Conference on*. IEEE, 2012, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6211780

[42] N. Y. Song, Y. J. Yu, W. Shin, H. Eom, and H. Y. Yeom, "Low-latency memory-mapped i/o for data-intensive applications on fast storage devices." IEEE, Nov. 2012, pp. 766–770. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6495887

[43] B. Widrow, J. M. McCool, M. Larimore, and C. R. Johnson, "Stationary and nonstationary learning characteristics of the LMS adaptive filter," *Proceedings of the IEEE*, vol. 64, no. 8, pp. 1151–1162, 1976. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1454555

[44] "CMSIS-SVD: System view description." [Online]. Available: http://www.keil.com/pack/doc/CMSIS/SVD/html/index.html

[45] P. S. Foundation, "Python.org," Jun. 2014. [Online]. Available: https://www.python.org/

[46] J.-S. Chenard, "Linux userspace i/o as a mechanism for rapid hardware driver development," 2012.

[47] "MATLAB - the language of technical computing." [Online]. Available: http://www.mathworks.com/products/matlab/

[48] R. Nikoukhah and S. Steer, "SCICOS-a dynamic system builder and simulator," in *Computer-Aided Control System Design, 1996., Proceedings of the 1996 IEEE*

International Symposium on. IEEE, 1996, pp. 430–435. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=555330

[49] "Xcos - hybrid dynamic systems modeler and simulator | scilab professional partner." [Online]. Available: http://www.openeering.com/xcos

[50] F. El Guezar, H. Bouzahir, P. Acco, K. Afdel, and D. Fournier-Prunaret, "Modeling and simulation in scicos: A case study," in *Computational Intelligence and Intelligent Informatics, 2007. ISCIII'07. International Symposium on*. IEEE, 2007, pp. 105–110. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4218404

[51] D. Maclay, "Simulation gets into the loop," *IEE Review*, vol. 43, no. 3, pp. 109–112, May 1997.

[52] T. Hwang, J. Roh, K. Park, K. H. Lee, K.-W. Lee, S.-J. Lee, and Y.-J. Kim, "Development of HILS systems for active brake control systems." Busan: IEEE, Oct. 2006, pp. 4404 – 4408.

[53] J. M. Cho, D. H. Hwang, K. C. Lee, J. W. Jeon, D. Y. Park, Y. J. Kim, and J. S. Joh, "Design and implementation of HILS system for ABS ECU of commercial vehicles," in *Industrial Electronics, 2001. Proceedings. ISIE 2001. IEEE International Symposium on*, vol. 2. IEEE, 2001, pp. 1272–1277. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=931663

[54] W. Deng, Y. H. Lee, and A. Zhao, "Hardware-in-the-loop simulation for autonomous driving," in *Industrial Electronics, 2008. IECON 2008. 34th Annual Conference of IEEE*. IEEE, 2008, pp. 1742–1747. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4758217

[55] F. Abugchem, M. Short, and D. Xu, "An experimental HIL study on the jitter sensitivity of an adaptive control system," in *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*. IEEE, 2013, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6647944

[56] OpenCores, "Wishbone b4," 2010. [Online]. Available: http://cdn.opencores.org/downloads/wbspec_b4.pdf

[57] J. Park and P. C. Diniz, "An external memory interface for FPGA-based computing engines." in *FCCM*, 2001, pp. 267–268. [Online]. Available: http://www.isi.edu/~pedro/PUBLICATIONS/ParkDiniz.fccm2001.pdf

[58] C. E. Cummings, "Clock domain crossing (CDC) design & verification techniques using SystemVerilog," 2008.

[59] A. Corporation, "Design debugging using the SignalTap II logic analyzer," in *Quartus II Handbook Version 13.1*, vol. 3.

[60] Altera, "HPS FPGA AXI bridges," in *Cyclone V Device Handbook*. 101 Innovation Drive, San Jose, CA 95134: Altera, Feb. 2014, vol. 5.

[61] "NIOS II performance benchmarks," Nov. 2013. [Online]. Available: http://www.altera.com/literature/ds/ds_nios2_perf.pdf

[62] Altera, "Qsys interconnect," in *Quartus II Handbook*, Nov. 2013.

[63] H. J. Koch and H. Linutronix Gmb, "Userspace i/o drivers in a realtime context," in *The 13th Realtime Linux Workshop*, 2011. [Online]. Available: http://www.osadl.org/fileadmin/dam/rtlws/12/Koch.pdf

[64] L. Torvalds, "Re: [GIT PATCH] more driver core patches for 2.6.19," Dec. 2006. [Online]. Available: https://lkml.org/lkml/2006/12/13/228

[65] H. J. Koch, "UIO: user-space drivers," Jun. 2007. [Online]. Available: http://lwn.net/Articles/236880/

[66] C. Ross and W. Bohm, "Using FIFOs in hardware-software co-design for fpga based embedded systems," in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*. IEEE, 2004, pp. 318–319. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1364657

[67] OPAL-RT, "Power electronic laboratories solution RCP HIL system," 2012. [Online]. Available: http://www.opal-rt.com/new-product/op4500-simulator-rt-lab-rcp-hil-system

[68] L. Moss, H. Gurard, D. Dare, and D. Bois, "Recent experience on an ESL framework for rapid design exploration using hardware-softweare codesign for ARM based FPGAs," 2012.

[69] F. S. Foundation, "Gnu radio," Jun. 2014. [Online]. Available: http://gnuradio.org/redmine/projects/gnuradio/wiki

[70] G. L. Bernocchi, G.-C. Cardarilli, A. Del Re, A. Nannarelli, and M. Re, "Low-power adaptive filter based on RNS components," in *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*. IEEE, 2007, pp. 3211–3214. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4253362

[71] M. Association, "Modelica and the modelica association," Sep. 2014. [Online]. Available: https://www.modelica.org/

[72] C. Brooks, E. A. Lee, and S. Tripakis, "Exploring models of computation with ptolemy II," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2010 IEEE/ACM/IFIP International Conference on*. IEEE, 2010, pp. 331–332. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5751519

[73] STMicroelectronics, "STM32f4 reference manual," Sep. 2013.

[74] ChibiOS, "ChibiOS/RT homepage [ChibiOS/RT free embedded RTOS]." [Online]. Available: http://www.chibios.org/dokuwiki/doku.php

[75] K. Mansley, S. Goldschmidt, and A. Dunkels, "lwIP - a lightweight TCP/IP stack - summary [savannah]." [Online]. Available: http://savannah.nongnu.org/projects/lwip/

[76] W. Yu, D. I. Wilson, J. Currie, and B. R. Young, "The robustness of PI and PID controllers in the presence of sampling jitter," *Chemical Process Control VIII, FOCAPO/CPC*, 2012. [Online]. Available: http://www.nt.ntnu.no/users/skoge/prost/proceedings/cpc8-focapo-2012/data/papers/004.pdf

[77] A. v. Balakrishnan, "On the problem of time jitter in sampling," *Information Theory, IRE Transactions on*, vol. 8, no. 3, pp. 226–236, 1962. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1057717

[78] G. Combs, "Wireshark." [Online]. Available: http://www.wireshark.org/

[79] F. Abugchem, M. Short, and D. Xu, "A test facility for experimental HIL analysis of industrial embedded control systems." Krakow: IEEE, Sep. 2012, pp. 1–4.

[80] M. Short and C. Cox, "RTE-SIM: A simple, low-cost and flexible environment to support the teaching of real-time and embedded control," *International Journal of Electrical Engineering Education*, vol. 48, pp. 339–358, Oct. 2011.

[81] Intel and Altera, "Altera and intel extend manufacturing partnership to include development of multi-die devices," Mar. 2014. [Online]. Available: http://newsroom.intel.com/community/intel_newsroom/blog/2014/03/26/altera-and-intel-extend-manufacturing-partnership-to-include-development-of-multi-die-devices

[82] B. Nahill, "PyCortexMDebug," 2013. [Online]. Available: https://github.com/bnahill/PyCortexMDebug

[83] S. Behnel, "lxml - processing XML and HTML with python," Jun. 2014. [Online]. Available: http://lxml.de/

[84] Altera and ARM, "ARM development studio 5 (DS-5) altera edition toolkit," Jun. 2014. [Online]. Available: http://www.altera.com/devices/processor/arm/cortex-a9/software/proc-arm-development-suite-5.html

[85] ARM, "Keil MDK-ARM," Jun. 2014. [Online]. Available: http://www.keil.com/arm/mdk.asp

[86] Xilinx, "ChipScope pro and the serial i/o toolkit." [Online]. Available: http://www.xilinx.com/tools/cspro.htm

[87] Z. Poulos, Y.-S. Yang, J. Anderson, A. Veneris, and B. Le, "Leveraging reconfigurability to raise productivity in FPGA functional debug," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2012, pp. 292–295. [Online]. Available: http://dl.acm.org/citation.cfm?id=2492781

[88] Y. Pang, K. Radecka, and Z. Zilic, "Optimization of imprecise circuits represented by taylor series and real-valued polynomials," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 8, pp. 1177–1190, Aug. 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5512688

[89] K. Radecka and Z. Zilic, "Arithmetic transforms for compositions of sequential and imprecise datapaths," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1382–1391, Jul. 2006. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1634633

[90] K. Radecka and Z.Zilic, "Using arithmetic transform for verification of datapath circuits via error modeling," in *VLSI Test Symposium, 2000. Proceedings. 18th IEEE*, 2000, pp. 271–277. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=843855