Is High-Level Design Representation Worthwhile?

by

Jason Hannula
B.Sc., University of Victoria, 2002

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Supervisor: Dr. M. Serra

ABSTRACT

As the complexity of computing systems increases, a unified design representation is needed to increase the level of abstraction in line with system level specification. Automation tools are required to achieve abstraction and are considered integral to the design representation. The ideal characteristics of a unified representation and its tool support are used to assess the currently available design representations.

The Handel-C language was chosen as a high-level design representations for further evaluation. It is the only current representation which meets the requirements of complete language implementation in either software or hardware and a mature automated tool set. As Handel-C is derived from ANSI-C, its implementation in software is equivalent. The ability to generate hardware compared to VHDL is assessed.

Handel-C produces prototype hardware implementations on par with VHDL. It surpasses VHDL with its simplicity in representing algorithms. Handel-C can replace low-level design for system prototyping without significant sacrifice.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This work presents the argument for moving towards high-level system design and attempts to assess whether this is possible with the current design representations and automation tools.

The underlying technologies and trends in industry and academia point towards greater integration of configurable hardware components in the design and implementation of computing systems. This flexibility demands an equivalent flexibility from design representations and methodologies at the system level. It also requires an increase in the level of design abstraction to contend with the increased complexity of system representations. The paradigm of Hardware Software Co-design attempts to bring the separate design paths for hardware and software under a single closely tied methodology. Unifying the design representations allows for exploration of both design spaces from the same specification with an easily movable partition.

A unified design representation allows the specification to be implemented in hardware, software, or a bit of both. The implementation must be possible from the same representation to all three possible targets through an automated process. This permits design space exploration in hardware, software, and the interface between them. Ideally, the design representation must map to both software and hardware without additional translation or constraint and provide an interface between the two.

An increase in the abstraction level associated with the migration to high-level design must be accompanied by design automation tools to handle the details hidden by the ab-

straction. It is the automation tool set and its ability to handle the abstraction, as much as the design representation itself, which determines the viability of moving to a high-level of system design. A representation without automated implementation remains only a representation and does not accomplish the goals of abstraction.

With these requirements, the currently available high-level design representations are assessed. At the time of writing, SystemVerilog [37][39], SpecC [21][28], SystemC [22][35], and Handel-C [4][16] are available as candidate representations. SystemVerilog provides support to software simulation but remains primarily a hardware definition and simulation language. SpecC is a modelling language which addresses the interface as well as software and hardware with waning support and no implementation tools. SystemC borrows from SpecC in its modelling aspects and leverages the power of C++ by adding libraries to describe hardware functionality. However, it is used primarily for testbench development and modelling. Only subsets of the language can be implemented in hardware limiting the migration of functionality between hardware and software. Handel-C is a constrained variant of ANSI-C with specific constructs for defining hardware functionality. Handel-C was chosen for further assessment because it met the criteria of full implementation in hardware or software and a mature automated tool set. Handel-C does not provide interface modelling as part of the language and this aspect of development required considerable manual design and implementation.

A testbench framework and the accompanying interface were developed prior to further assessment of Handel-C. This required integrating a testbench application, the host platform's operating system, the PCI bus, a PCI interface on the Field Programmable Gate Array (FPGA), and the experiment implemented on the FPGA. Design automation was provided for the development of the device driver but the interface sections between the device driver and the testbench application and the PCI interface and the experiment were designed and implemented manually.

Three algorithms,

- Tiny Encryption Algorithm [41],

- Discrete Cosine Transform for JPEG compression [34], and
- Coordinate Rotation Digital Computer [12]

are implemented to assess the abilities of Handel-C versus a standard hardware implementation language, VHDL. The development experiences and final implementation metrics are compared to determine the value of Handel-C as a unified design representation.

Chapter 2 presents the context and a conceptual framework for the work presented. It provides motivation for the discussion of high level system representations through Hardware Software Co-design and the underlying technologies, Electronic Design Automation, Configurable Computers, and Field Programmable Gate Arrays which are influencing the current trends in design.

In Chapter 3, the argument for adopting a unified design representation is given. This is followed by a survey of currently available representations. The representations are assessed according to the requirements presented in the first part of the chapter and a unified design representation is chosen for further experimentation.

Development of the interface and testbench are documented in the initial sections of Chapter 4. This is followed by descriptions of the experiments performed including commentary on the implementation process for each language. The chapter closes with a discussion of the experimental results.

Chapter 5 summarizes conclusions drawn from these comparative implementations and provides suggestions for future work.

Chapter 2

Context and Conceptual Framework

This Chapter is an introduction to the areas of:

- Hardware Software Co-Design,
- Electronic Design Automation Tools,
- Configurable Computers, and
- Field Programmable Gate Arrays.

In total, the current trends in these areas are changing the ways in which computing system are designed. Increasing complexity, especially in embedded systems and system-on-a-chip, is pressuring system modelling and design methodologies towards greater levels of abstraction. The context for these trends provides a basis of understanding for both the design method changes projected and their evaluation later in this work.

2.1 Hardware Software Co-Design

A formal design methodology or flow is the sequence of steps followed during a design process. Employing an appropriate and precise design flow is considered essential for building systems which work correctly, safely, and meet external constraints such as cost, time, performance, or quality. Several flow models have been proposed in software engineering that are also used in the larger area of system design [42]. These include the *spiral model*, *extreme programming*, and *successive refinement*. The often referred to *waterfall model* is

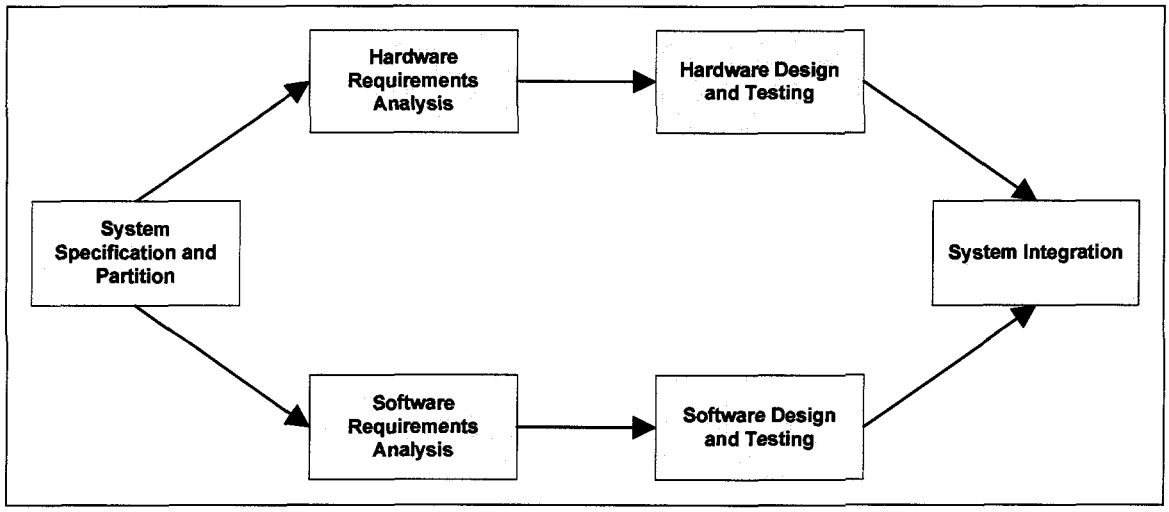


Figure 2.1. *Conventional System Design*

usually dismissed as unrealistic, requiring complete specification of system requirements before the process begins. The models follow similar paths where initial prototypes or subsystems are specified, designed, built, and tested leading to enhancements or refinements and further iterations of the process.

Many systems have both hardware and software components. During the design and build phases these components are traditionally treated independently as shown in Figure 2.1 [42][38], with the design paths diverging early in the process. This is often done without a formalized method to determine the best design partition [38]. There is little or no interaction during the individual design phases, as independent teams or different contractors are usually employed for each component. Integration of the components is towards the end of the process and the design partition is not necessarily investigated as part of design space exploration.

The term hardware/software co-design refers to the methodology, tools, and practices that support the integration of hardware and software components during system design and development [27]. Hardware/software co-design is distinguished from other design methodologies in the following ways:

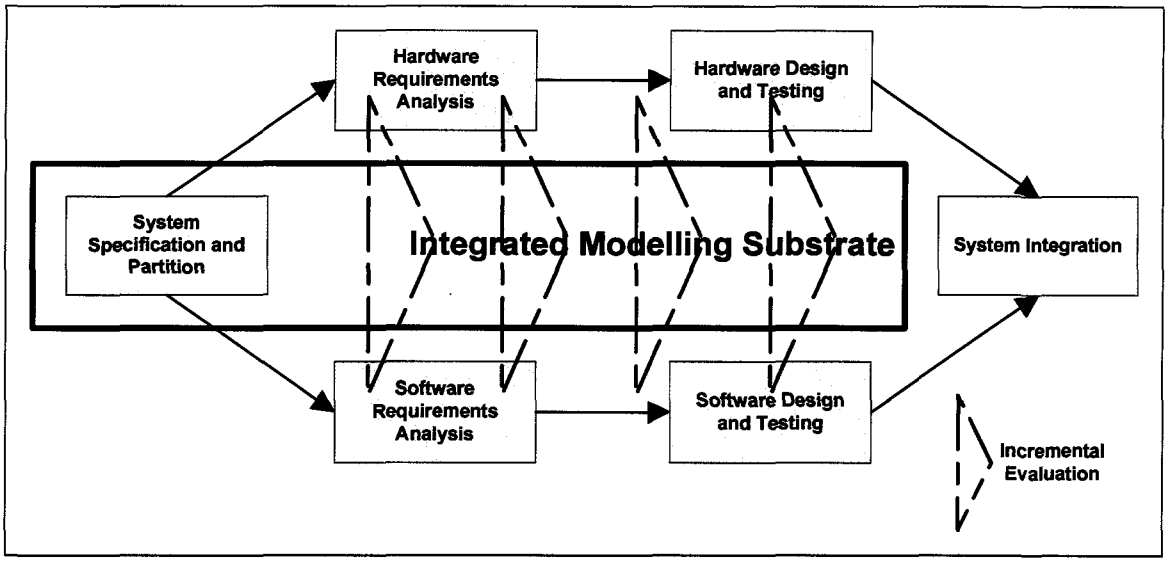


Figure 2.2. *Unified System Design Environment*

- cooperative design of hardware and software components,
- unification of the separate hardware and software design paths,
- free movement of functionality between hardware and software during design exploration, and
- meeting system-level objectives by exploiting the synergism of hardware and software through their concurrent design.

This leads to more flexible design strategies, where hardware and software designs proceed in parallel with feedback and interaction at the center of the process [38].

Figure 2.2 is a graphical representation of this unified design environment. The key differences between this model and the model presented in Figure 2.1 is the emphasis on an integrated modelling substrate. Support for a unified design environment requires more than the occasional meetings between hardware and software design teams. In Figure 2.2's idealized view, the modelling substrate minimally provides an integrated database for representing the hardware and software components, and the interface between. As well as a unified representation, tools that support *design space exploration* including movement of

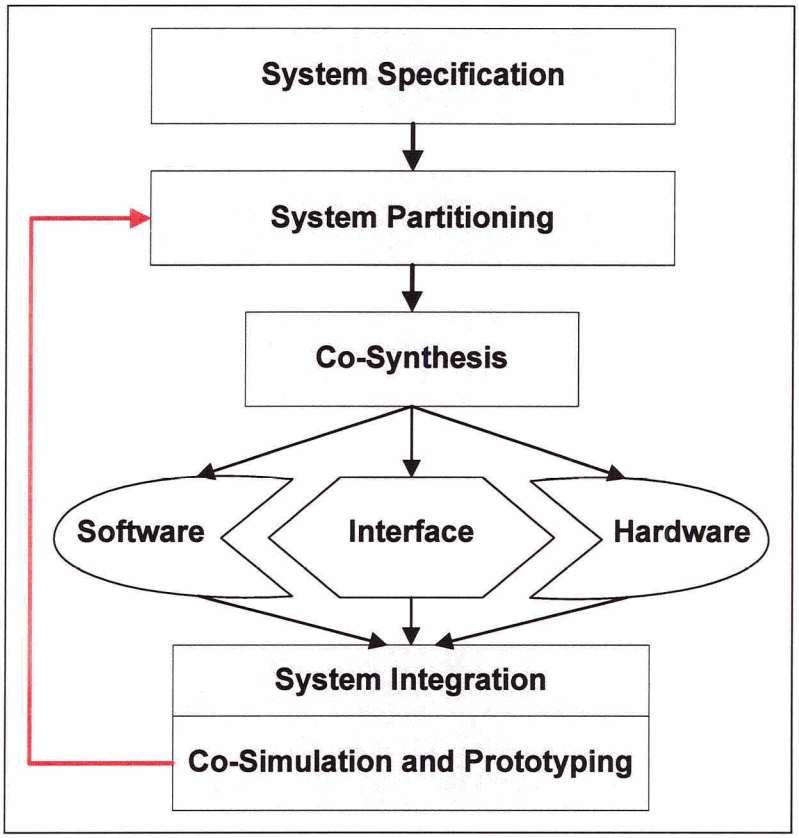


Figure 2.3. *Hardware Software Co-Design Process*

the system partitions, efficient and effective *co-simulation* platforms, rapid *evaluation* of changes in functionality, *prototyping* platforms, and *co-synthesis* for generating an implementation are needed [38].

Figure 2.3 is another view of the co-design process. Two crucial aspects of the process are the feedback represented by the red arrow, and the interface component. In simple system design, the interface is assumed from the original partition. This can cause difficulty at the time of integration or during the segregated design flow. The interface is explicitly shown because exploration of the partition changes the interface implementation. Implicit in the process is a unified system representation to varying degrees. It supports the movement from system description to hardware, software, and interface representations throughout design iterations and to a final implementation. This unified representation supports

exploration of the partitioned design space.

2.2 Electronic Design Automation Tools

Computer-aided design (CAD) and computer-aided manufacturing began in the 1960's. CAD uses the computer's interactive graphics capabilities and processing power to automate simple repetitive tasks. The designer can then focus on a higher level of representation or abstraction. It initially allowed designers to move away from the tedious tasks associated with developing designs on paper and progressed to include analytical support, simulation, and simultaneous design access for team based development [17].

Electronic design automation (EDA) is an extension of CAD specific to integrated circuit design and generally features high levels of automation. Initially, integrated circuit designers could focus on handcrafting every transistor because manufacturing cost and chip performance were the only constraints. As the complexity of designs have increased, it has become infeasible to increase the size of design teams and length of design schedules to compensate. Design automation has been employed to insulate the designers from the lowest representational levels [18].

The design flow for circuits generally moves from higher to lower levels of abstraction. The stages can be characterized as behavioural, functional, structural, switch-level, and geometric. The associated design representations and some of the EDA tools are summarized in Table 2.1. The exact tool set varies based on the specific supplier. The level of automation increases as the abstraction decreases. At lower levels, the tools provide complete automation for labour intensive, high data volume, repetitive tasks such as place and route. Little automation is offered for high levels of design. Tasks, such as architectural design in VHDL, are usually more creative in nature and based both on inspiration and experience [18].

Abstraction Level	Circuit and EDA Characteristics	
	Circuit Representation	EDA Tools
Behavioural	VHDL, Verilog	Textual Design Entry, Synthesis
Functional	RTL	Functional Simulation and Verification
Structural	Logic Gates	Schematic Design, Logical Simulation
Switch-level	Transistors	Timing Analysis, Power Analysis
Geometric	Physical Layout	Place and Route, Floorplanning

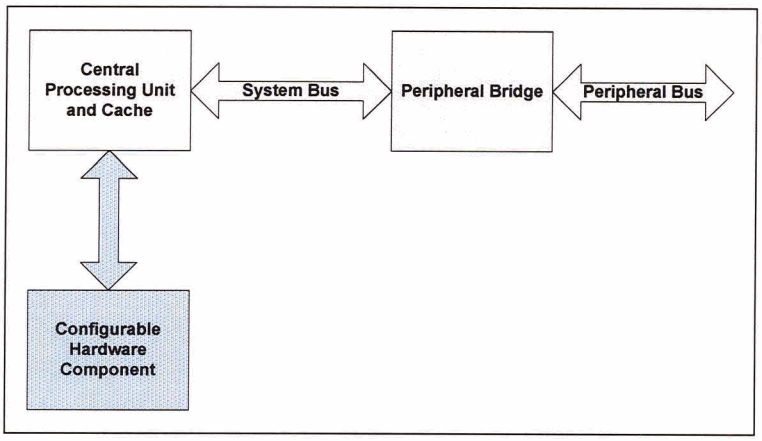
Table 2.1. *Circuit Design Abstractions and Representations*

2.3 Configurable Computers

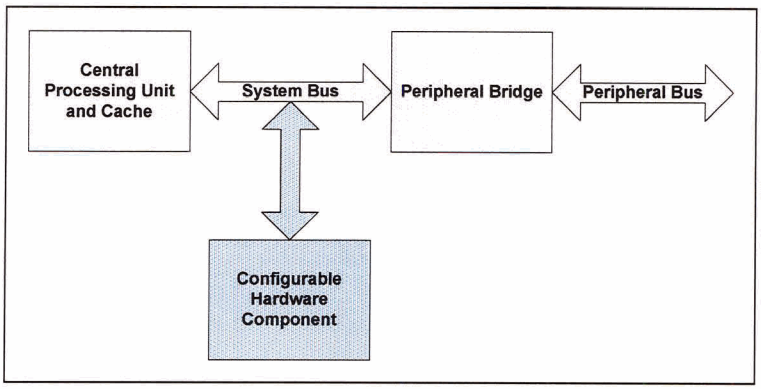
Configurable computers are computing machines which use configurable hardware architectures. Configurable architectures are usually a mix of fixed, general-purpose hardware components and configurable, user-defined hardware components. The general-purpose components can range from general-purpose computer systems to processor cores. The user-defined components are typically one or more programmable logic device which can be configured for specific functions. The architectures are classified by how close the two components are organized in a communication sense.

There are three architecture classifications: (a) *reconfigurable processing unit*, (b) *tightly-coupled*, and (c) *loosely-coupled*. They are shown schematically in Figure 2.4. In (a), the reconfigurable processing unit is coupled at an instruction level with the fixed and reconfigurable components directly connected. Tightly-coupled architectures in (b) connect the components through a system bus structure, while loosely-coupled systems in (c) rely on a peripheral bus for communication between the components [13]. The classifications are not limited to board level implementations, but include system on chip implementations especially when processor cores are used as fixed components.

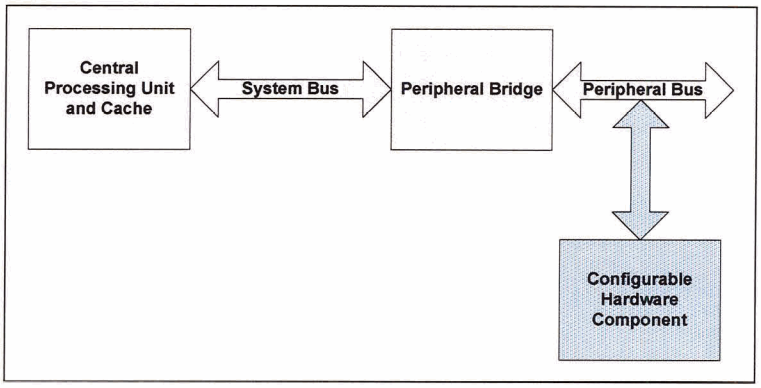
Classification of the configurable components comes from the specific technology employed. The configurable taxonomy used in publications tends to be inconsistent between



(a) Reconfigurable Processing Unit



(b) Tightly-Coupled



(c) Loosely-Coupled

Figure 2.4. *Configurable Computer Architectures*

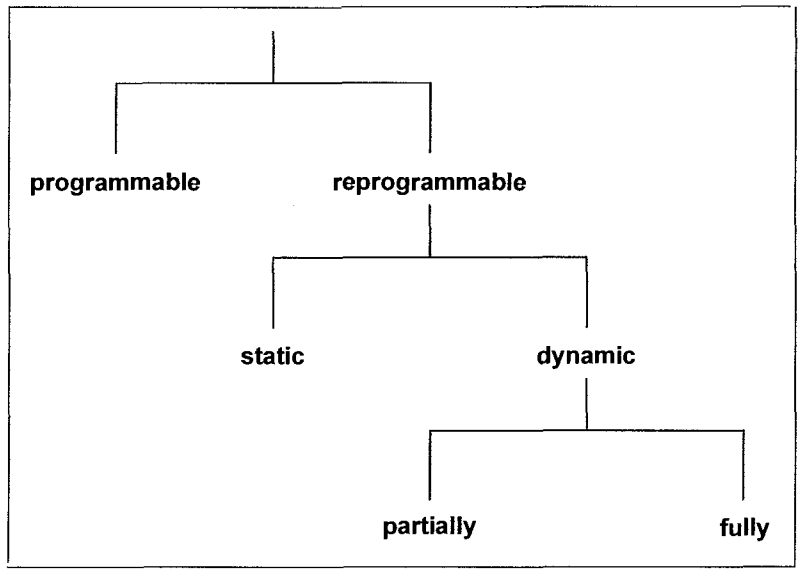


Figure 2.5. *Taxonomy of Configurable Components*

authors. Terms such as reconfigurable, configurable, and dynamic configuration are not clearly defined and are occasionally used interchangeably. Figure 2.5 presents a hierarchal taxonomy consisting of the following three concepts:

- programmable versus reprogrammable,
- static versus dynamic, and
- partially versus fully.

For the remainder of this work the terms configure and program will be used interchangeably. *Reprogrammable* components can be configured with a new design more than once during its lifetime, while *programmable* components are configured only once. Currently most reprogrammable or reconfigurable devices fall into the category of *static* (offline) configuration. This requires the component to be isolated from the running system during its configuration cycle. *Dynamic* (online) configuration would allow the reconfiguration to occur while the system is in operation. In this case, a running application could reconfigure circuitry to meet requirements of the current task [31]. *Partially* configurable components allow a portion of the current configuration to remain when it is programmed

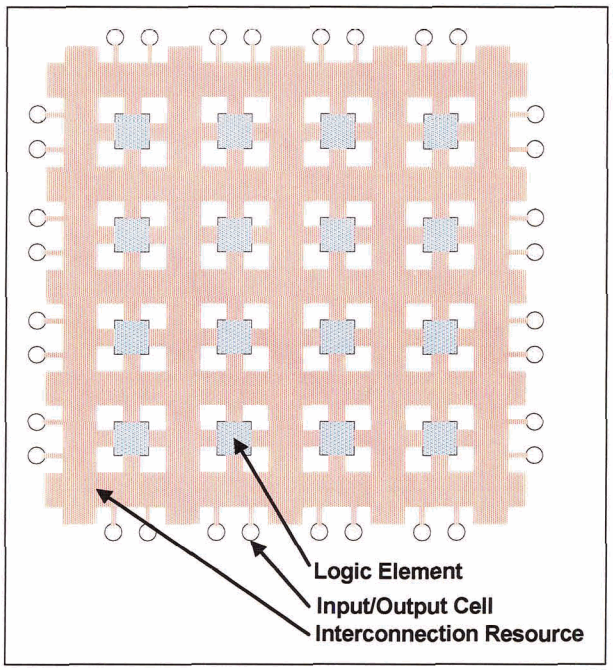


Figure 2.6. *General FPGA Architecture*

with a new configuration. *Fully* configurable components require configuration of the entire device.

2.4 Field Programmable Gate Arrays

A programmable logic device is an integrated chip that is programmed with a digital circuit design by a user [15]. This is unlike application specific chips (ASIC's), which are fabricated with a single static design. Programmable logic devices range from the simple, such as Programmable Logic Arrays, to high density devices such as Field Programmable Gate Arrays (FPGA's), which can implement sequential circuits and memory. Specialized FPGA's are available which may include additional embedded memory, fast multipliers for digital signal processing, or embedded micro controllers and processors [10][11].

The basic FPGA architecture consists of an array of elemental logic structures that is connected by user-defined routing to each other or input/output cells. Figure 2.6 shows this

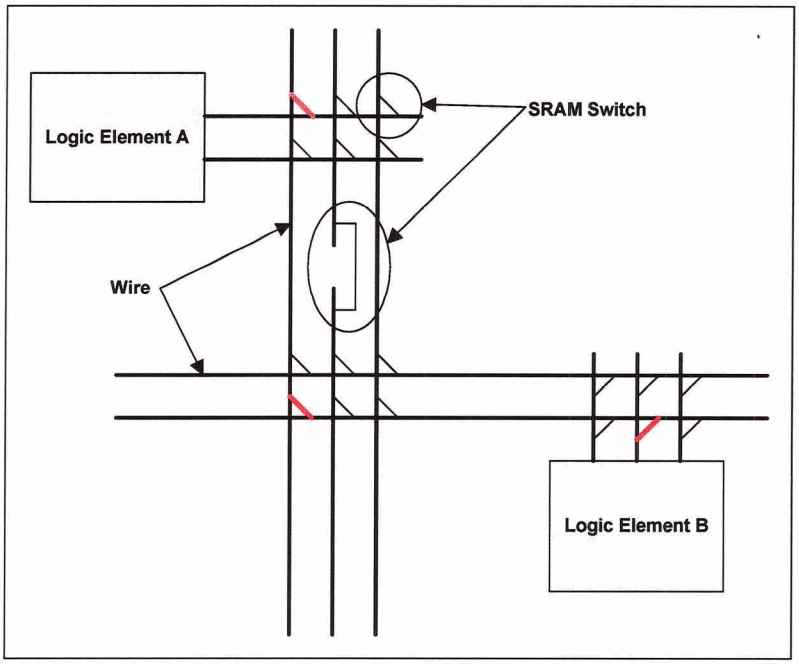


Figure 2.7. *User-Defined Routing Structure and Switches*

generalized architecture. It employs a regular matrix of logic blocks (blue on the figure) with a grid of interconnection resources (red on the figure) surrounded by input/output cells [15], which are the FPGA's connection to the exterior world.

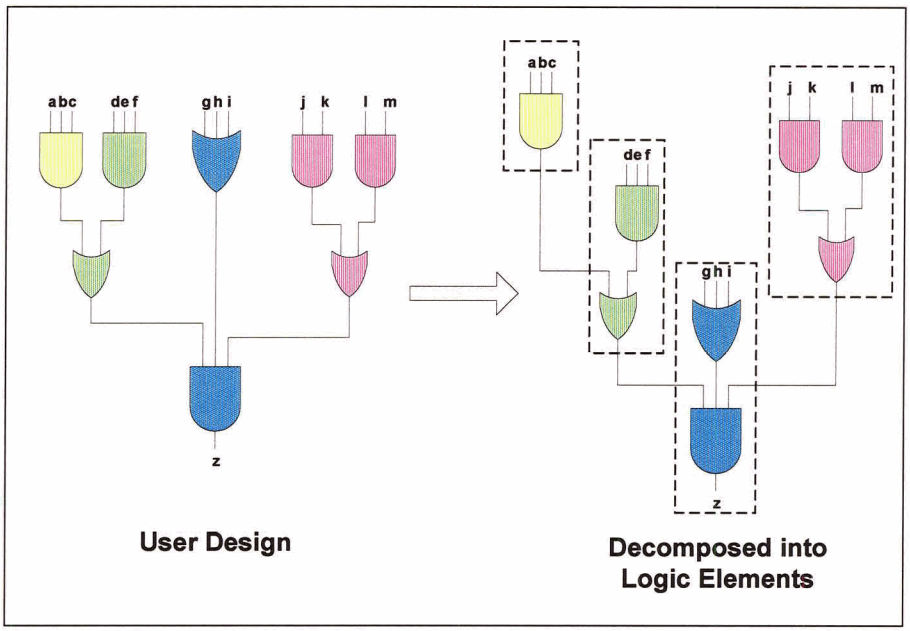
An elementary FPGA logic block is usually based on look-up tables and memory components. Such a logic element can be programmed to implement any boolean function based on the number of input and output lines. For example, the Altera Apex logic element employs a four input look-up table. The look-up table can implement any single output boolean function with up to four input variables. The logic block can also implement a sequential circuit by using its associated memory components to capture its sense of state.

The routing structure is a matrix of wires with switchable connections. The connections are defined by the user when the chip is programmed. The wires vary in length and layout depending on the specific manufacturer. Some layouts favour column-long interconnections running from one side of the chip to the other, while others emphasize many short paths. The switches generally are of two types: static ram (SRAM), completely

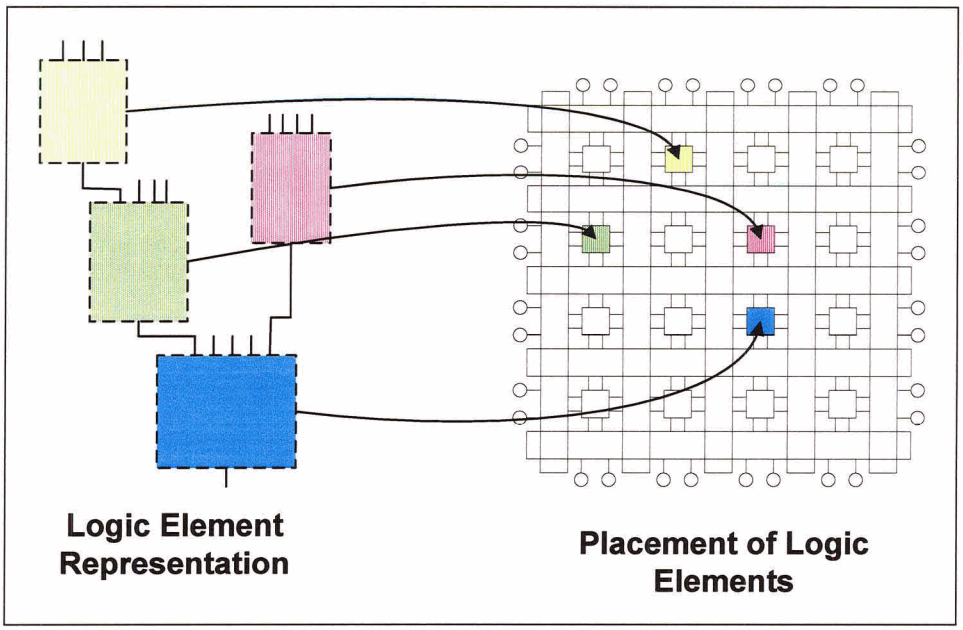
re-programmable, or anti-fuse, programmable only once. The specific technology and arrangement details are usually proprietary. Pathways between logic elements are assigned by setting specific switches [14]. User definition of the structure is accomplished through the programmability of switches. Figure 2.7 shows an arbitrary SRAM based routing structure with both long and short wires. By setting the red switches on, a path is created from the output of Logic Element A to an input of Logic Element B.

An FPGA is programmed or configured with a user design by mapping the design to the programmable logic blocks and routing resources. This is called the "place and route" function. Figure 2.8(a) shows the logical representation of the user design being decomposed into blocks which can be implemented by individual look-up tables. The decomposed blocks are then assigned to, or "placed" in, specific logic elements on the FPGA as shown in Figure 2.8(b). A mapping of the logical connections between the placed blocks and the routing resources is then created. This is the "routing" function. The design's inputs and outputs are routed from available input/output cells to the correct logic blocks. This mapping process is usually iterative to ensure constraints, such as timing, are not violated. When an acceptable place and route is found, a programming file is created. The programming file defines the configurations for look-up tables and the state of switches in the routing resources [24].

The programming of an FPGA is generally performed with the assistance of EDA tools (see Section 2.2). After a design has been developed, a logical representation is generated by the tool set. Its decomposition into logic elements is supported by data structures such as Binary Decision Diagrams. These data structure provide a framework for manipulation of logical expressions [19]. The wealth of knowledge found in graph theory supports "place and route" functions.



(a) Decomposition of User Design to Logic Elements



(b) Placement in Logic Elements

Figure 2.8. *Place Function*

Chapter 3

An Argument for Change in Design Language and Tools

This chapter brings together the ideas introduced in the last chapter with speculation about the future of embedded computing to define the requirements for a practical unified system specification required in co-design. The requirements will form the basis for evaluating a number of unified specification systems available at the time of writing.

The overwhelming majority of digital devices currently produced are destined for embedded devices. A home may have one or two desktop computers and between 35 and 50 embedded systems. These range from simple devices such as household systems (temperature controllers, washing machines, microwaves) to the complex like modern multifunction hand-held devices (cell phones with personal digital assistants and digital cameras) [40]. Increasingly, the trend in this market is to incorporate programmable logic to handle acceleration of algorithms rather than develop custom logic in the form of ASIC's. It is an economically driven change caused by the steadily increasing cost of custom chip manufacturing. In contrast, programmable logic devices like FPGA's, have increased and increasing capabilities in clock speed, embedded memory, and gate equivalency without the non-recurring engineering costs of a custom logic solutions [24]. Additionally, the consumer electronics market continues to increase the rate at which improved products are released. The incorporation of programmable logic can help maintain legacy systems by offering field programmability to upgrade or correct functionality. In total these influences

and trends point towards an increasing number of configurable computing platforms.

Other factors which may drive the increased use of configurable computing are seen in the trends predicted by Reiner Hartenstein in his invited paper at the 9th IEEE International Conference on Electronics, Circuits and Systems in 2002 [26]. Hartenstein foresees a movement away from the traditional Von Nuemann architectures in computing devices. The standard fetch-decode-execute cycle causes a bottleneck in data and computation throughput enforcing strictly sequential execution of an algorithm. He anticipates a movement towards architectures with run-time reconfigurable datapaths and parallel executions were the hardware is the algorithm and it configures according to the data's requirements. This requires the adoption of configurable computing platforms to provide hardware flexibility.

Designs which incorporate configurable computing platforms highlight the drawbacks of traditional design methods discussed in Section 2.1. The lack of unified system representations limits design space exploration especially concerning hardware-software partitioning. Design representations exist on a continuum, with pure modelling system and pure implementation language at the extremes. In reality, most representations fall between these two with strong tendencies to one or the other.

Modelling systems such as finite state machines, dataflow diagrams, or statecharts provide independence from implementation [20]. This gives the designer the flexibility to explore design ideas based on data flow or control flow depending on the design focus. Models usually require an additional translation to become an implementation in either hardware or software. The dataflow for a mathematical operation can be modelled with a dataflow diagram as in Figure 3.1. This gives a representation of how the information can be organized in time sequence between components. In most systems, this can not be accepted as a design input and must be translated to an implementation focused language.

Implementation languages such as C, VHDL, or Java can be compiled directly into machine code, configuration files, or simulated for testing and verification. They tend to lack pictorial representations in their language constructs and require translation to make

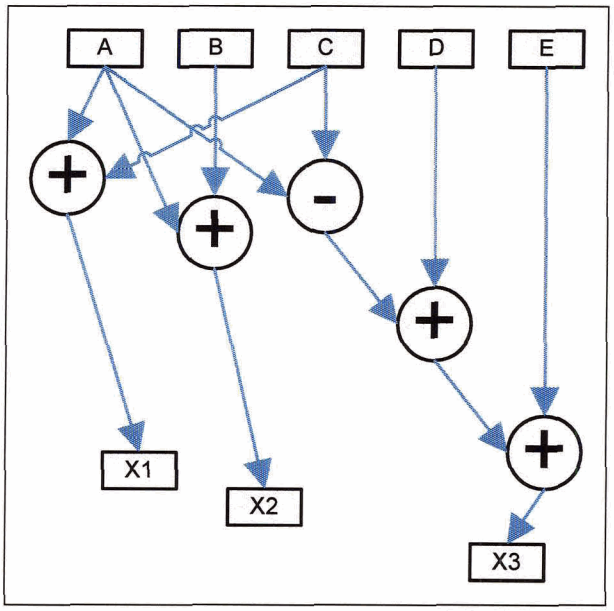


Figure 3.1. *Modelling Language Example: Dataflow Diagram*

design visualization possible. Figure 3.2 is a snippet of C code from the Tiny Encryption Algorithm. It is a cyclic application of shifts and exclusive-or's to encrypt two word of information. This is a good representation for arithmetic functions, but for the untrained reader the implemented is not clear.

Ideally a unified design representation would combine the visualization of a modelling system with the compilation capability of an implementation language. An additional requirement for a unified design representation in the context of co-design, is the ability to generate implementations for hardware, software, and an interface from the same specification. This facilitates exploring changes in the partition without changing the specification.

A crucial point is that, for a design representation to be useful, it must be supported by EDA tools. An automated process for compiling the design into hardware configurations or executable software codes does not exist unless there is some tool support. This favours the use of implementation languages. A modelling system which is not supported by EDA tools will require hand translation and coding into an implementation language. This break in the automated process can potentially introduce errors. It also makes the use of iteration

```
while (n -- > 0) {
        sum += delta;
        y += (z<<4)+k[0] ^ z+sum ^ (z>>5)+k[1] ;
        z += (y<<4)+k[2] ^ y+sum ^ (y>>5)+k[3] ;
}
v[0] = y;
v[1] = z;
```

Figure 3.2. *Implementation Language Example: C Code*

difficult in design exploration, a key stage in most design methodologies. Software and hardware simulations or implementations for verification can not be easily reached from the modelling stage. The iterative cycling between verification and design specification is broken if the modelling system can not be implemented through automation. This reduces the success and applicability of refinement or prototype based design methodologies.

Another factor to be considered is the current practice in industry. Developing a new system which is unfamiliar for designers will not be easily embraced. The current skill set of designers needs to be exploited, not thrown away. Maintenance of legacy systems, designs, and intellectual products also influence a new unified design representation. A company with extensive legacy designs will not switch to a new representation if the legacy systems can not be incorporated easily. The level of existing design knowledge will limit, in practise, the adoption of an entirely new or unfamiliar systems. Wayne Wolf, in *The Computer Engineering Handbook* [36], describes embedded systems including common design process. The system specification process is detailed as follows:

> "In practice, many systems are specified in the C programming language. Many practical systems combine control and data operations, making it difficult to use one language that is specialized for any type of description. Algorithm designers generally want to prototype their algorithms and verify them through experimentation; as a result, an executable program generally exists

> as the golden standard with which the implementation must conform. This is especially true when the product's capabilities are defined by standards committees, which typically generate one or more reference implementations, usually in C. Once a working piece of code exists in C, there is little incentive to rewrite it in a different specification language; however, the C specification is generally a long way from an implementation. Algorithm designs are usually written for uniprocessors and ignore many aspects of I/O, whereas embedded systems must perform real-time I/O and often distribute tasks among several precessing elements. Algorithm designers often do not optimize their code for any particular platform, and their code is certainly not optimized for any particular embedded platform. As a result, a C language specification often requires substantial re-engineering before it can be used in an embedded system [43]."

Ideally, a unified design representation would provide:

- Visualization,
- Simulation,
- Compilation,
- Implementation, and
- Integration of Existing Representations.

This is at the system level in a "CO"-sense for all three components of the design, hardware, software, and the interface. The representation and supporting tool set treats the design as a single entity in a unified development environment rather than independent streams with ad hoc interface exploration.

There are three possible directions of note at the time of writing for developing a unified system representation:

- building from the C language,
- building from the existing hardware specification standards VHDL and Verilog, and
- adding implementation to modelling system.

As noted in Wolf's description of embedded system specification, C has become the de facto standard for system specification based on algorithms. The language requires the addition of hardware modelling and specification to become a unified specification. VHDL and Verilog are the industry and IEEE standards for hardware specification [1][2]. They have grown out of the need to develop hardware description at a levels of abstraction above transistors as noted in Section 2.2. These languages are not usually used for describing algorithms and require extensions to generate software implementations. Modelling languages, as described above, generally lack EDA tool support for the generation of implementation in either hardware or software. SystemC [22][35], SpecC [21][28], Handel-C [4][16], and SystemVerilog [37][39] are the system representations which are receiving the majority of interest at the time of writing.

3.1 Survey of System Representations

SystemVerilog is a recently ratified hardware description and verification language standard built upon the existing VHDL and Verilog hardware description languages. It is intended to provide support to SystemC for simulating hardware from a system-level descriptions. It only supports software implementations through the standardization of allowable data types with the C language. There is strong support for this standard in the EDA community. Software can not be generated from this language, instead co-simulation is achieved by linking to a system model such as SystemC. Without the ability to explore the entire design and partition options, SystemVerilog does not meet the needs of a unified system representation but can be a key support framework for other representations [39].

SpecC is primarily a modelling language which has been developed at the University of California, Irvine. It is based on StateCharts [25] and SpecCharts [33] and is also a form of hierarchical and concurrent state diagrams. The terminal behaviours for the state diagrams are described in implementation specific languages. One of the major advancements in SpecC is the abstraction of communication channels between components. This

is the only surveyed system representation which specifically addresses the interface between components with abstraction or implementation. However, the support of SpecC has fallen since its inception and there does not exist any EDA tool support for generation of implementations in software or hardware.

SystemC is a collection of libraries for the C++ language that extends it to encompass ideas from hardware such as concurrency, ports, sensitivity, and channels. Hierarchical concepts already exist because of the object oriented nature of C++. The language has borrowed extensively from SpecC especially in the modelling of communication through channels. SystemC is seen as the primary system modelling language. There is EDA support for simulation of both hardware and software generated by the C++ libraries. However, there is no limit on using any aspect of C++. As a result, the language is consider only partially synthesizable. SystemC is seen as a model for testbench generation and system simulation, but does not provide the completely automated flow from system specification to implementation in hardware and software.

Handel-C is a modified form of C which does not allow floating point data types and limits the use of pointers and recursion, but includes constructs for the generation of hardware specifications. It provides language elements for parallel execution, ports, channels, and memories. With minor adjustments, most algorithms developed in C can be implemented in hardware from Handel-C using an EDA tool stream. It is possible to link Handel-C algorithms to existing hardware implementations expressed in VHDL, Verilog, or EDIF formats. Handel-C code can be used to generate software to run on a hardware implementation which has been defined with Handel-C as well.

At this point, Handel-C is the best possible candidate for direct evaluation. It provides an EDA tool stream targeting FPGA technology and leverages existing knowledge of C programming and algorithms. Handel-C is seen as a starting point for a unified system representation. However, Handel-C does not provide a model or abstraction of the interface between hardware and software in a co-design sense. The entire language can be implemented as strictly hardware or strictly software. A partition of functionality between

hardware and software can be made but the partitioning does not result in an interface implementation. The interface remains a design specific, hand tooled, implementation.

The idea of interface in SpecC and SystemC is broken into interfaces, ports, and channels which abstract the ideas of operations, the object which operations occur over, and protocol for communication, respectively [22]. This gives a clear abstraction of the interface components and allows for IP based implementation for different channel protocols. In future, any design representation should include the ability to explicitly model the interface.

It is good to step back and consider what other concerns there are with regards to industry accepting a new unified system representation. A recent panel discussion at MEMOCODE '03 titled: *"Should the space of implementation possibilities be determined by the abilities of high-level synthesis and validation?"* included the section "Example Issues and Problems". It highlights some of the current open questions for research and industry. Some of it is quoted below:

> "What must designers give up to take advantage of synthesis or validation in a given high-level design framework? What and how much does a designer have to gain by adopting a high-level design framework?
> What low-level control over implementation details are designers most willing to trade for increased ease-of-design and productivity? How does a designer decide how much performance (in terms of speed, power, size) is worth sacrificing for moving to a high-level design framework [23]?"

These questions form a good basis for evaluating a design representation. What is needed is some design metrics to determine the scale of "sacrifice" a designer must make. The speculative question about "how much sacrifice" a designer is willing to make in moving to high-level design is worth pursuing. The question can be inverted to determine how much sacrifice a particular representation requires. Then, depending on the situation the appropriate representation can then be chosen. In the move from assembly language level programming to compiled language programming, the sacrifice for most cases was consid-

ered worthwhile. In cases where coding efficiency beyond the capabilities of a compiler are required, software designers fall back on assembly language programming for small subsets of code. The same design style can be applied for system-level design.

To evaluate the sacrifices a designer must make to use Handel-C as a unified design representation, there are two aspects of the sacrifice to consider, software and hardware. Handel-C software implementations can be handled with existing C language compilers and will produce reasonably equivalent implementations. From a software perspective, there is no sacrifice in using Handel-C instead of C. There are no existing commercial hardware compilers which take the complete ANSI-C language as an design input. To evaluate the efficiency of a Handel-C hardware implementation it needs to be evaluated relative to a current standard hardware design representation.

Chapter 4

Experiment and Results

The experiments performed for this work were designed to determine the extend of sacrifice or gain a hardware designer experiences by using a unified design representation like Handel-C. Only one published article by Mylonas, Holding, and Blow was found showing a comparison between a hardware implementation developed using Handel-C and a standard hardware description language [32]. The authors implemented a DES cipher block using Handel-C and its tool set targeting a Xilinx FPGA and compared the results to a VHDL implementation on the same FPGA. The article discussed resource usage (number of Xilinx CLB slices) and the maximum clock speed determined by longest path in the implementation. It found that the Handel-C implementation achieved greater clock speeds (74.145MHz versus 68.999MHz), but required approximately 50% more resources (3,813 versus 2,524 CLB slices). There were no result regarding design time for either implementation or the extent of training of the designers in either language.

The experiments presented in this chapter provide an extension to the published work by documenting the design time for each version of the implementation as well as resource usage and maximum clock speed. Additionally, commentary and reflections on the experience of designing in both paradigms in light of the arguments presented in Chapter 3, conclude the chapter. Three different projects are implemented:

- the Tiny Encryption Algorithm [41],
- a variation of the JPEG compression [34], and
- a CORDIC algorithm [12].

These experiments were chosen because they represent some of the application areas which Morris describes in the Computer Engineering Handbook for successful implementation in reconfigurable processors [31]. The full list of areas is as follows:

- image processing,
- cryptography,
- database and text searching,
- compression, and
- signal processing.

These areas exploit common characteristics such as parallelism, low storage requirements, "decision-free" processing patterns, local data paths, and integer arithmetic.

Before the experiments can begin, an interface between the testbench software running on the host system and the experiments implemented in hardware needs to be developed. As discussed in Section 3.1, Handel-C does not model the interface between hardware and software. The interface development remains a system and design specific, hand tooled process. Section 4.1 documents the development process undertaken for the experimental platform used.

4.1 Development Platform

4.1.1 Development Hardware

All of the hardware design and implementation for this work was performed on a Windows 2000 workstation equipped with a 1.8 GHz Intel Xeon processor and 1 Giga-Byte of RAM. The workstation uses an Altera Apex 20KC1000 peripheral component interconnect (PCI) Development Board to implement hardware designs. Figure 4.1 shows a schematic view of the workstation and the development board. This board supports 32 bit and 64 bit PCI communication at both 33 and 66 MHz. It provides an Apex 20KC1000 FPGA

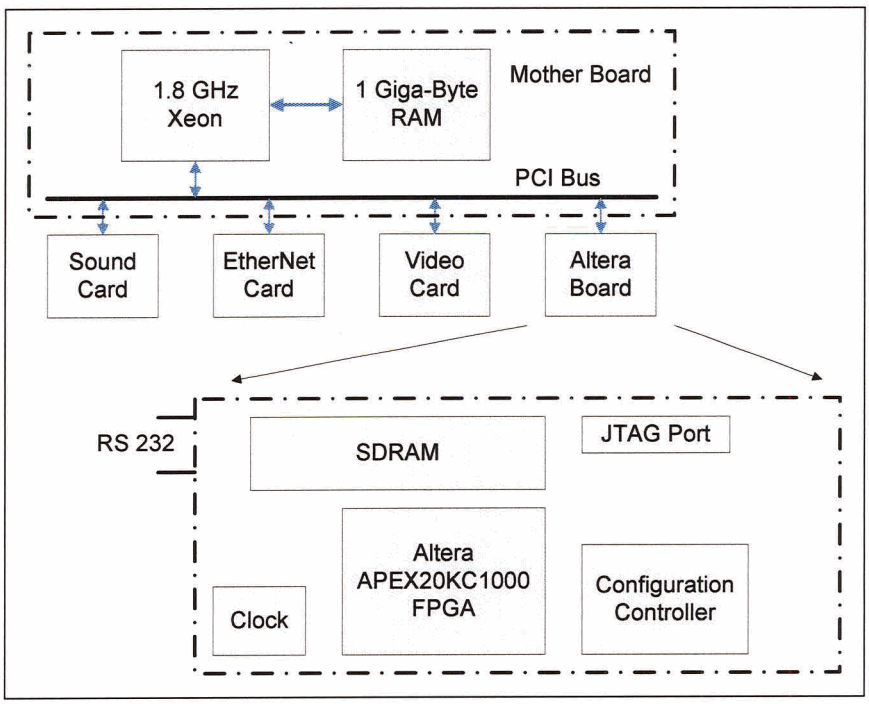


Figure 4.1. *Development Hardware Schematic*

from Altera, SDRAM modules, a reconfiguration controller, expansion slots, and external communication ports [6]. Altera donated the MegaCore MT64 parameterized PCI communication functional block intellectual property to assist in the development of projects for the development board [7].

4.1.2 Development Software

Three separate software suites were used during hardware development: Altera's Quartus II design suite, Celoxica's DK design suite, and Jungo's WinDriver suite.

Altera's Quartus II design software is used in conjunction with the Altera PCI development board. It provides a design entry, synthesis, place-and-route, verification and FPGA programming environment. Designs can be entered either as text files using VHDL, Verilog, or EDIF formats, or as schematic layouts. The schematic design tools provide access to parameterized intellectual property functions such as multiplexers, adders, or FIFO

buffer structures. These have been developed by Altera to target specific technologies on its FPGA's. The software synthesizes the user design to a form which can be mapped to the FPGA resources with the place and route tools. The FPGA on the development board is programmed using the Quartus II software through the development board's JTAG port [8][9].

Celoxica's DK development suite supports development of hardware designs in Handel-C. Handel-C is derivative of the C programming language with additional language features for specifying parallel processes, hardware connections, and clock functions [4]. Designs developed in Handel-C are compiled to a hardware specification in the EDIF format [3]. The EDIF specifications are exported to Quartus II for further processing and programming the FPGA.

Before new hardware can communicate with an operating system, the operating system must have some basic information regarding the hardware. This information is usually contained in a device driver. The device driver performs two main functions: device configuration and initialization, and management of data movement. The driver encapsulates this information so an application can treat the device as an abstraction. Developing device drivers is not a simple process. If a hardware device changes, the device driver must reflect these changes. In this work, the reconfiguration of hardware is central to the experiments. To limit the time taken to develop drivers during this work, an automated software tool, Jungo's WinDriver was used. WinDriver detects the new hardware and generates the device drivers and library functions in the C language to handle communication between the host system and new hardware [29]. It also contains a hardware debugger to test a PCI connected device. It is a GUI interface for reading and writing to addresses mapped to the device. The hardware and communication library can be tested before any software is developed.

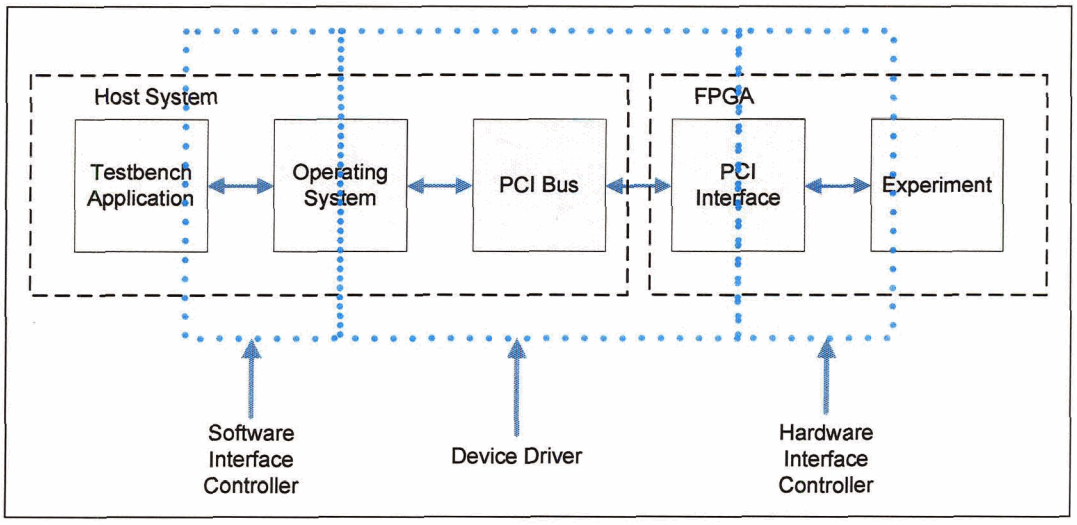


Figure 4.2. *Interface Design Schematic*

4.1.3 Interface

The system configuration discussed in Section 4.1.1 requires an interface between the host system software and the FPGA based hardware experiments. Figure 4.2 shows this complex implementation. It relies on the correct interaction between a software testbench application, the host platform's operating system, the PCI bus, a target PCI interface implemented on the FPGA, and the individual experiment implement on the FPGA. Existing intellectual properties and development applications were used extensively in the interface development to assist and automate the process where possible.

An FPGA based PCI interface intellectual property was donated by Altera for use with the Altera PCI Development board. This assisted in encapsulating the interaction between the PCI bus and the FPGA.

Developing device drivers for an operating system to communicate with hardware devices is one of the most difficult tasks in system development [30]. With the FPGA PCI interface implemented on the FPGA, the Jungo WinDriver development application was employed to assist in the automation of device driver development. This provides the link between the operating system through the PCI bus to the FPGA based PCI interface. The

development application also provides basic communication functions for further software interface development.

Two sections of the interface remain for hand design and coding.

To ensure consistency for the experiment implementations, a common interface between software and hardware was developed. This removes the need to custom develop an interface solution for each experiment. However, there is a requirement for custom development in the more general case usage of Handel-C. The visible ends of the common interface consists of an FPGA hardware component and a host system software component. The hardware component was developed in VHDL using the Quartus II design environment. The software component is built using the library functions generated by the WinDriver software.

The PCI interface provided by Altera is incorporated into the hardware component and handles the target-side of PCI communications. An additional control and buffering structure is added between the PCI interface and the experiment to simplify communication protocols for the experiment. To decouple the experiment's clock from the PCI clock, a handshake protocol is used. This allows communications between the PCI interface and the experiment to occur in two different clock domains. Reset and buffer clearing functionality is incorporated as part of the hardware controller.

The software side of the interface acts in the master capacity for PCI communications with the hardware as its target. This portion of the interface is added to the experiment's software testbench as a C language library. For each experiment, appropriate data is sent over the PCI bus to verify the functionality of the hardware implementation. The software and hardware development must be aware of each other. The interface only facilitates data transfer it does not manipulate the data.

4.1.3.1 Hardware Controller Development

The testbench controller was designed in two phases, structural design for the dataflow path, followed with the development of a control structure model using a finite state ma-

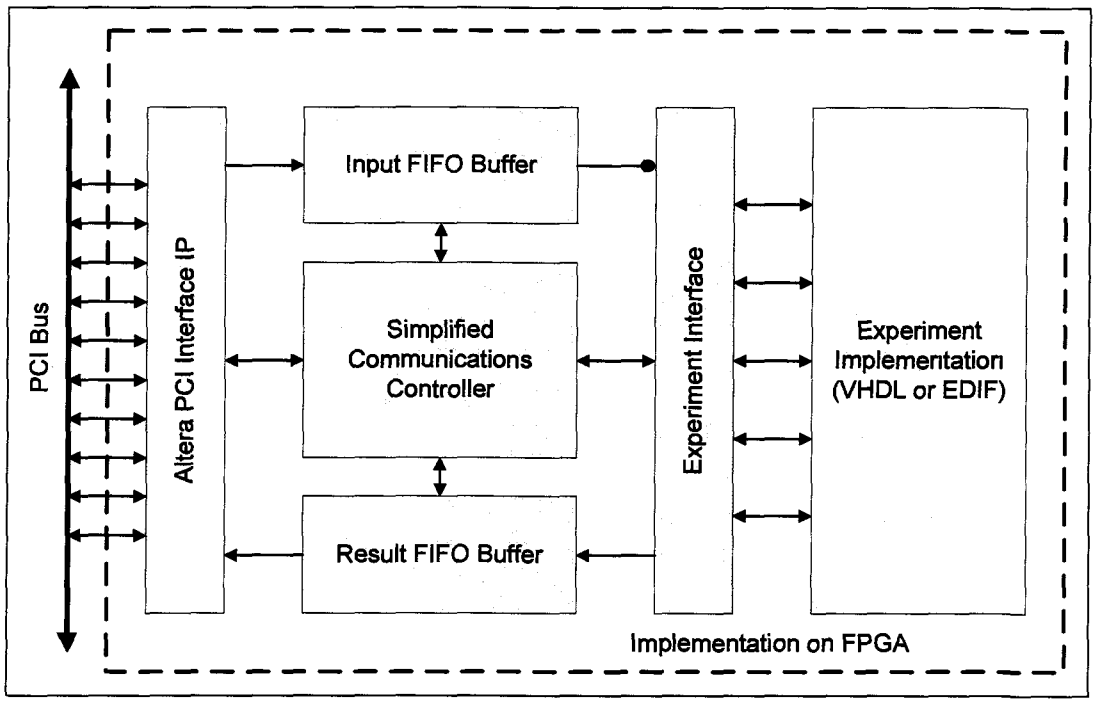


Figure 4.3. *Hardware Controller Schematic*

chine. The dataflow path was constructed using the Quartus II schematic design tool. It was then converted to VHDL. The control structure was coded in a behavioural VHDL case construct. The complete VHDL code listing is included as Appendix A.

A schematic of the controller's structure is shown in Figure 4.3. It consists of two FIFO buffers, the Altera PCI Communication IP, a simplified testbench controller, and standardized interface for each experiment to implement.

The control model is built from a central ready state which waits for requests from the PCI interface or the backend design. Only one of these requests can be serviced at a time. The PCI communications are given priority. Figure 4.4 is the simplified finite state machine for describing the controller behaviour. Servicing a request from either the PCI interface or the experiment requires setting the correct datapath and control signals. By using timing diagrams provided in the Altera documentation [7], the appropriate control signals are asserted to service read and write requests from the host system. The controller

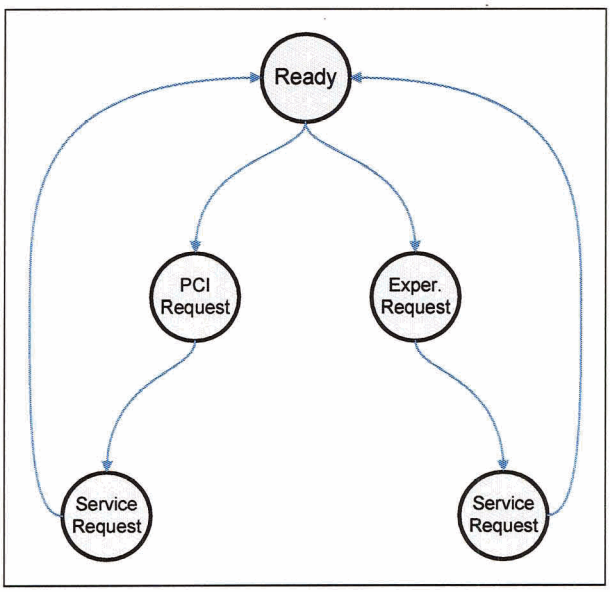


Figure 4.4. *Simplified Hardware Controller Finite State Machine*

is tied to the PCI clock to simplify the timing sequences.

Communication for the backend designs are isolated from the PCI protocols. A simplified handshake protocol is implement for data transfers to and from the experiment using three control lines each. The same protocol is used for both communication directions. The controller operates the ready and received lines with the experiment asserting the request line. The input protocol is mapped in Figure 4.5. It shows the three control lines labelled ready, request, and received and the data bus. The ready lines are background signals from the controller, the input ready indicates the presence of data in the input buffer and the result ready indicating that there exists open slots in the results buffer. The backend design asserts the appropriate request line to indicate a transaction request. When a data request is acknowledged by the controller valid data is placed on the input bus and the received signal is asserted. Upon receipt of this signal and capture of data, the backend design deasserts its request signal.

System reset functions are implemented to allow the controller, buffers, and backend to enter a known state. Four options are implemented, final data pop, input buffer flush,

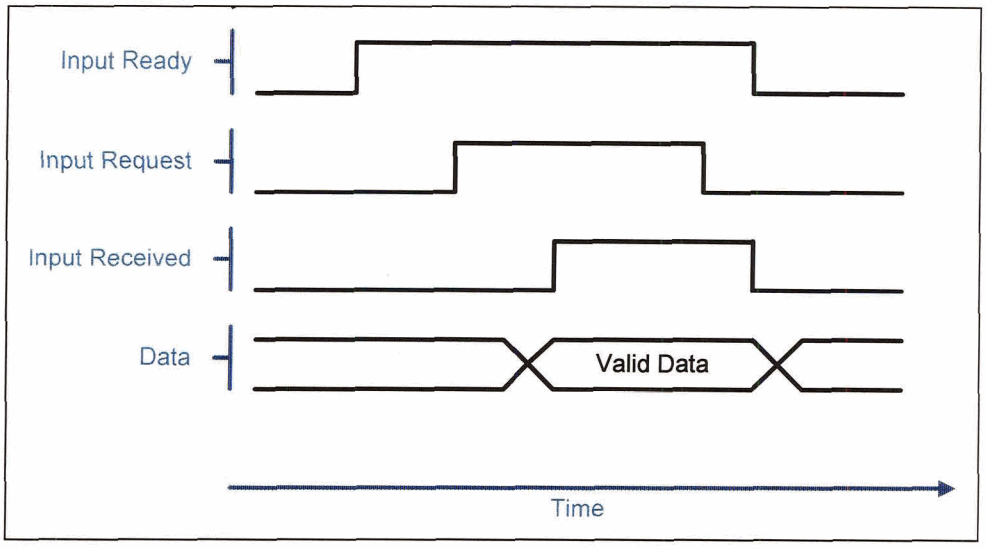


Figure 4.5. *Input Transfer Handshake Protocol*

results buffer flush, and a system reset which flushes both buffers and sends a reset signal to the backend design. The final data pop is required because result from the controller are offset by one with a padding zero result as the initial read value after a system reset.

4.1.3.2 Host System View

The host system is able to view the hardware implementation via the PCI bus. The PCI interface is implemented with three 32-bit registers mapped to system addresses. Two of the addresses are write only and are used for data transfer and the reset signals. The other is read only and is used for retrieving returned results. The design implements polling to retrieve data. The PCI interface returns a null value (0xFFFFFF) to indicate there are no results available. Because this bit pattern may in some instances be valid data, a watchdog timer must be implemented in the software testbench to ensure a valid result of 0xFFFFFF is captured. The write function does not return any indication that the write was successful or aborted (if the input data buffer is full a PCI abort signal is returned by the controller). These limitations force the host system software designer to be aware of the hardware implementation such as the buffer size and data requirements per result. These are reasonable

constraints because the testbench interface is intended to exercise the backend design's functionality not for general use.

4.1.4 Verifying the Testbench

A simple average function is implemented as the initial design to ensure the testbench is functioning correctly. The design receives two numbers as binary input from the software testbench via the PCI bus. These values are added and the sum is shifted right by one digit. This final result is then returned to the software through the PCI bus again.

This design was chosen for its simplicity to reduce the possibility of design error occurring in the hardware design. From the hardware perspective, this exercises the controller's ability to manage multiple requests from both the PCI and the backend design. The design also tests the ability of host software to manage a mismatch in the number of data inputs required for an expected output result, in this case two inputs per output.

Testing was performed in three stages. First, the controller and backend design were simulated without the PCI interface. This allowed the design to be debugged in isolation from the PCI protocol.

The second stage was a hardware test employing the WinDriver hardware debugger. The PCI interface, controller, and backend design were synthesized as a single design. This was used to generate a programming file for configuring the Altera Development Board. The Windriver software was used to produce the necessary operating system drivers. This phase was used to ensure the PCI communications were interacting with the backend design correctly. Specifically, the buffer full and empty conditions were checked and the reset options. One aspect of the testbench which must be managed in software is the one slot delay of data in a result stream.

The third stage of testing built upon the C library generated by the WinDriver software. A simple program was developed which read data from a file, sent the data via the PCI bus to the test design, and output the returned results to the screen.

4.2 Experiments

There are three stages of development for each experiment, testbench development, Handel-C hardware development, and VHDL hardware development. In line with Wolf's standard development process discussed in Chapter 3, a C implementation of each experiment is developed first as a testbench to verify the hardware implementations. A partition choice is made regarding which functionality to implement as hardware and the interface software is inserted in exchange for the corresponding C function.

Each experiment is incorporated as the backend of the hardware component. For experiments in VHDL, this is done entirely in the Quartus II environment. The VHDL design is synthesized along with the hardware component to a netlist format. The DK suite is used to design experiments in Handel-C. These are then compiled to a EDIF file which is also a netlist format. The EDIF file is imported to the testbench in Quartus II and is combined with a synthesis of the hardware component.

At this point, each experiment, regardless of initial design format, undergoes the same automated process. The netlist representation is used by the place-and-route function to generate a possible assignment of logic elements and interconnection resources on the FPGA. Upon successful generation of a resource mapping, the appropriate configuration file is created. Programming of the FPGA is performed by the Quartus II software using the configuration file.

Once the experiments have been developed, the implementations are verified with simple examples. Simple cases are used for verification because the experiments are not intended to meet commercial standards nor to be competitive as hardware accelerators. The development is targeted at proof of concept and prototyping rather than production. In a commercial setting, the designs would be profiled to determine where further refinements are to be made prior to verifying the design tolerances.

4.2.1 Tiny Encryption Algorithm

The tiny encryption algorithm (TEA) was developed by David Wheeler and Roger Needham at Cambridge University in 1994 [41]. The encryption relies on a large number of iterations to achieve a reasonable level of encryption. It encodes two 32 bit numbers using a 128 bit key and an additive factor delta, which is derived from $(\sqrt{5}-1)2^{31}$. Two 32 bit encodings are returned. The order and pairing of encoding must be preserved for correct decoding.

4.2.1.1 Software Implementation

The C language listing for the TEA is found in Appendix B. It consists of functions for opening the interface connection, encoding two data words, decoding two data words, and a main testbench function. In the encoding function, the original software only implementation is commented out and the interface communication is substituted. This maintains the original function call structure. Decoding is performed in software to ensure the hardware implementation is functioning correctly. The main function is used to exercise the hardware by calling the encoding routine and decoding the return results. The initial, coded, and decoded values are printed to the screen to verify the hardware's correctness.

4.2.1.2 Hardware Implementation

The TEA was implemented in Handel-C first. The translation to Handel-C was straight forward for the small segment of code that makes up the encryption algorithm because language constructs exist for shifts and exclusive-or. Appendix C lists the Handel-C code used to generate the final implementation. Code to handle communications with the hardware controller make up the majority of the code. This implementation hardcodes the key which is functionally similar to entering the key in a write-only memory to ensure security. No attempts were made to manually optimize the implementation and any optimization of the final netlist is provided by the DK compiler.

The VHDL implementation involved reducing the C code to a data structure and an appropriate control structure. The code listing is included as Appendix D. Each expression and assignment in the algorithm is used to construct a data path with registers to hold the intermediate values. The control structure, in the form of a finite state machine, handles communication with the hardware interface and processing the input data. This is also a naive implementation which does not attempt optimizations such as pipelining. The Quartus parameterized functions where used extensively to generate data path components.

The majority of development time for each implementation was used to resolve bugs with the initial versions. The largest source of errors in the VHDL version came from matching communication, register, and counter enable signals with the correct state and the clock. In the Handel-C version, debug time was used to refine communication with the hardware interface. In both cases, as they are the initial projects, the novice status of the developer contributed to the length of development time.

4.2.2 JPEG Compression

The JPEG image compression standard [34] employs a three step process consisting of:

- Discrete Cosine Transformation (DCT),
- coefficient quantization, and
- encoding.

The DCT converts the spatial representation of the image (Cartesian location and intensity) to spectral representation just as a Fast Fourier Transform converts a time-intensity signal to frequency-magnitude representation. Coefficient quantization reduces the amount of information required to represent the image by reducing representational precision. This step causes the lossy nature of JPEG compression. The last step encodes the transformed and quantized information taking advantage of the large number of zero's present in its representation. The encoding uses a code scheme such as Huffman codes or arithmetic codes for non-zero coefficients and run-length encoding for the zero values.

4.2.2.1 Software Implementation

The software implementation and testbench was developed using base code provided by Dr. Nigel Horspool of the University of Victoria. Standard JPEG compression uses an eight by eight pixel block for processing. In this implementation, it has been reduced to a four by four block to be more manageable in the testing environment given the hardware interface employs FIFO buffers of depth 16. The code provided for reading image files accepts eight-bit grayscale bitmap images. Verification of the implementations was performed using bitmap files with random geometric shapes. The image files are visually inspected from before and after compression to ensure correct functionality. Appendix E contains the C code listing for implementing the software version of and the function hardEncode which implements the software interface for performing the DCT in hardware. The function call for the software version of DCT has been commented out in the compression function and hardEncode is called instead.

4.2.2.2 Hardware Implementation

The DCT function was implemented in hardware because it is the most likely candidate for acceleration. Quantization requires a division for each matrix element which can be easily coded for software but the hardware implementation of division is considered best accomplished by IP functions or simplified operations. Encoding and saving the transformed and quantized information to a file can not be accomplished using the interface developed for these experiments. The DCT function requires implementation of matrix multiplication which is known to exercise hardware compilation and synthesis tool and their ability to meet timing or size requirements. This is meant as a challenge for the Handel-C compiler.

Accuracy for the DCT calculations require precision beyond integer number representations and floating point implementations are not available in either VHDL or Handel-C. A fixed point representation of eight bits was chosen for the implementations. In the hardEncode function in Appendix E, data is multiplied by 256, shifting it left by eight bits, before

it is sent to the hardware. The hardware returns rounded signed integers to the software.

The JPEG DCT was implemented in Handel-C initially as a direct translation from the existing C code. This presented the first translation problem. In Handel-C, all variables must have their size declared. The loop control variables were set at the size of two bits because loops run from zero to three indexing the matrix elements. In the C implementation, the loops are controlled by the condition less than four such as:

$$for\,(i = 0;\, i < 4;\, i++)$$

In Handel-C, if the loop control variable i is set at two bits it will always be less than four and this structure is an infinite loop. All of the for-loops from the C version were converted to do/while structures in Handel-C.

The remaining development time was dedicated to determining why correct code would not generate functionally correct hardware. The compilation process from Handel-C to hardware introduced the source of incorrect functionality. Language documentation for Handel-C warns that implementations of multiplication and division are not tuned for increasing clock performance. All experiment implementations use the PCI clock as a reference clock but are not designated as critical paths for timing analysis by the Quartus II place and route tools. The initial Handel-C implementations could not meet the timing requirements dictated by the PCI clock because of a naive implementation of multiplication by the DK compiler. Before this was discovered a number of other possible solutions were attempted including the structure found in Appendix F, the final implementation. The two loop sections, one for row multiplication and one for column multiplication, were changed to a single row multiplication loop and a matrix transpose function. It was thought that the optimizations were missing the differences in the loop structures. The solution to the incorrect behaviour was to introduce a clock division factor in the Handel-C clock definition.

Implementation of the DCT in VHDL required substantially more development time in the initial design of data paths and the looping control structure needed for loading the data from the hardware interface into the FPGA's RAM based memory structures. Once this

was accomplished, designing the FSM for controlling matrix multiplication did not present many difficulties. The Quartus II software provides multiplier implementations that have been optimized either for area or speed. The speed optimized implementation was chosen after the design lessons learned during the Handel-C implementation.

4.2.3 CORDIC Function

The COordinate Rotation DIgital Computer (CORDIC) algorithm [12] is an iterative solution method for trigonometric and transcendental functions using only shifts and adds. It is derived from the general rotation transform:

$$x' = x\cos\phi - y\sin\phi$$

$$y' = y\cos\phi + x\sin\phi$$

which rotates a vector in a Cartesian plane by angle ϕ. It can be used in two modes, rotation or vectoring. In rotation mode, the input vector is rotated by a specified angle. In the vectoring mode, the input vector is rotated to the x axis and the angle required is recorded. These are accomplished by three equations:

$$x_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-i}$$

$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i}$$

$$z_{i+1} = z_i - d_i \cdot \tan^{-1}\left(2^{-i}\right)$$

where

$$d_i = -1 \; if \; z_i < 0, \; +1 \; otherwise$$

in rotation mode, and

$$d_i = -1 \; if \; y_i >= 0, \; +1 \; otherwise$$

in vectoring mode.

4.2.3.1 Software Implementation

The software implementation, included as Appendix H for the CORDIC algorithm is simplified to the rotation mode only. It implements the arctan factors in an array in anticipation that the hardware will use a ROM implementation for the same factors. The iterations are limited to 16 giving about three decimal places of accuracy. For verification the software prompts the user to enter an angle between 0 and 90 degrees. The sine and cosine are calculated by the CORDIC algorithm and output to the screen. Degrees were chosen instead of radians to simplify the entry of test angles. Implementation in radians would only differ in the arctan values in the lookup table.

4.2.3.2 Hardware Implementation

As with the JPEG experiments, fixed point representations were used for the CORDIC hardware. In this case, 12 bits of precision were used requiring the input data to be multiplied by 4096 before being sent to the hardware.

The VHDL implementation was undertaken first because data flow schematic were provided in the reference work [12]. As suggested for the iterative implementation, a ROM was used to store the arctan values as a simple look-up table. VHDL supports arithmetic shift operators since its 1993 revision, however these are not support by the Quartus II software. This required the development of a custom shift component to handle arithmetic right shifts of a 32-bit number by an input amount ranging from 0 to 15. The remainder of the VHDL development focused on creating the control finite state machine. The final code in found in Appendix J.

For the Handel-C implementation included in Appendix I, a translation from the original C implementation was used. With the lessons learn from the two previous implementations, the development of Handel-C code took very little time. It was not anticipated that the addition and shift operations would create a divided clock requirement. This was an incorrect assumption and when all other possibilities were exhausted the clock was divided

by a factor of five to achieve correct functionality.

4.3 Results

Three metrics were recorded for each language version of the experiments:

- Development time, in hours,
- Resource usage, in logic elements and memory bits used, and
- Estimated maximum clock, in Mhz.

The results in this section must be considered in the context that the experiments were performed. The developer should be considered a novice in both languages used for the experiments. With each implementation, familiarity with the language and software tools, and the availability of existing designs for code reuse increased. The goal for each implementation was correct functionality for the small test cases as a first stage in prototyping or iterative design, not optimization for any other design constraint. In light of the fact that experiments were all performed by a single developer, they can not be considered statistically significant in the comparing design representations. These results can however be used for determining sample requirements for future experimentation using larger groups of developers leading to statistically significant comparison results. Commentary regarding the use each design language and development environment given in this section provides insight into what potentially influences the design process. This answers the questions posed about the nature of gain or sacrifice that a developer must consider regarding an appropriate design representation.

Table 4.1 summarizes the three implementation metrics for each language version of the experiments. Commentary and discussion for each metric is presented in the following sections.

Experiment	Language	Development Time	Resource Usage	Clock
TEA	Handel-C	2.5 Hours	514 LE's, 0 Mem	80.01 MHz
	VHDL	10.5 Hours	667 LE's, 0 Mem	71.49 MHz
JPEG	Handel-C	31 Hours	4147 LE's, 1024 Mem	29.80 MHz
	VHDL	45 Hours	3049 LE's, 1280 Mem	31.53 MHz
Cordic	Handel-C	4 Hours	1089 LE's, 512 Mem	61.66 MHz
	VHDL	3.5 Hours	911 LE's, 512 Mem	70.61 MHz

Table 4.1. *Summary of Implementation Metrics*

4.3.1 Development Time

The complexity of a design affects development time in two ways, the design effort required by the developer and the time required by EDA tools to generate a final place and routing for either simulation or hardware verification. For example, the Quartus II software required 7 minutes, 55 seconds to place and route the final VHDL version of TEA and 17 minutes, 9 seconds for the final VHDL JPEG implementation. In these two cases, the same number of debugging iterations for JPEG required about twice the time as for TEA.

Experience and design reuse are the greatest influence on the design effort needed by the developer. The differences in development time for the VHDL implementation of TEA and Cordic show this affect the best. Based on resource usage, Cordic is more complex than TEA, but the development time required for the Cordic was approximately 40 percent of the development time required for TEA. The number of debugging iterations required is influenced by experience as well. The use of a standardized interface for each experiment greatly simplified the location and nature of communication problems. The steep initial learning curves for both development environments and languages stabilize by the third experiment causing the development times to converge as compared to the first experiment. It is expected that additional experiments would continue with this same trend in development time.

A further refinement in the debugging process which will greatly decrease the number

of iterations in future is the inclusion of an accurately simulated clock in the simulation waveform input. The simulation waveform used for these experiments was set with an arbitrary clock period. The need to divide the clock for the Handel-C implementations of JPEG4 and Cordic would have been determined much quicker if the accurate clock had been used during their simulation.

All of these effects are captured by the development time metric. In this sense, the metric shows the interaction between the developer, the design representation, and the design tools. It is anticipated that development time would continue to decrease as a developer's experience increases. Because all of these different factors influence the development time as it was measured, the differences between the two design representation are not considered significant. The large difference for the TEA implementations should be attributed more to learning curve for the tools and debugging rather than the design representation.

As a general observation, the translation from C to Handel-C required very little time or effort after the basic difficulties with declared size variables were encountered once. The initial work to design in VHDL with a C specification required much more effort. During the debugging iterations of the development process, the low level controllability of VHDL assisted in quickly diagnosing the source of errors. Once the Handel-C code was determined to be correct, there much less accessibility to the internal design generated by the Handel-C compiler. Depending on the accuracy of the developer, this can stretch the debug time greatly for Handel-C designs.

4.3.2 Resource Usage

The resource usage metric reflects the needs of the experimental design without the hardware interface. After a verified implementation has been obtained, the implementation is analyzed in isolation as a stand alone design.

For these experiments, the target hardware technology has both configurable logic elements and memory bits [5]. Implementations which use memory bits for either RAM or ROM elements can reduce the number of logic elements that would be required for the

same design. This is true regardless of the design language choice as these designations are available in both. There are limitations on the access of memory bits during a single clock cycle because they require addressing rather than simply multiple wiring paths for the logic element only implementation.

The developer is not in direct control of the resource usage. Designs in either language undergo synthesis which determines the resources required. In the case of Handel-C, the compiler performs optimization before it maps the design to the target technology. In the Quartus II environment, the parameterized VHDL components have been optimized for the technology.

Resource usage can be used for estimating the minimum sized technology which can be targeted by future design refinements. In the case of a predetermined technology choice, this can frame the optimization effort required in the future.

The resource usage for each implementation is similar regardless of the design representation. In the JPEG implementations, there is a trade off of memory bits for logic elements which does not show advantage for either representation.

4.3.3 Estimated Maximum Clock

The clock metric was obtained from under the same circumstances as the resource usage. This ensures the estimate reflect the design and not the tightly constrained PCI interface. The metric is based on the longest register to register (input/output pins are considered registers) path measured as the total of logic element delays and routing delays. It assumes that all register to register communication must occur in one clock cycle.

This is not always a true assumption. In the case of the VHDL JPEG implementation, several clock cycles are allowed with null states for the propagation of multiplication results to the correct registers. The timing analysis is not aware of these design choices. It is not clear whether the same control can be exercised in the Handel-C environment. Optimization of the design may remove null statements which attempt to delay output capture.

It is not clear that the clock estimates for Handel-C designs with a divided clock are

truly accurate. An additional analysis for the Handel-C Cordic implementation without a divided clock was performed. This design is known to not function correctly. The clock estimate returned was 62.34 MHz. One of these estimates can not be accurate because the functioning implementation requires the clock to be divided by a factor of 5 to achieve functional correctness.

In general, the estimated maximum clock is a poor metric for determining the timing requirements of the design. To accurately assess the clock required, lower level information regarding critical paths for signal stability and actual clock cycle counts needed for setup of a signal. As an initial metric to determine the order of the design speed it is sufficient for estimating how much refinement is required during further design iterations. Given the skepticism about the accuracy of this metric, it is not possible to determine if one design representation creates significantly faster designs than another.

Chapter 5

Conclusions and Future Directions

The goal of the work presented is to argue for the adoption of high-level system design, moving away from separate lower-level hardware and software design paths. The initial chapters presented background material emphasizing the current trends in programmable logic and configurable computers as motivation for the migration towards high-level design representations. This led to the assessment of one of the currently available potential design representation, Handel-C in relation to hardware design practices. It has been emphasized that the design representation can not be separated from the design environment and automated design tools which support it. Therefore, the accompanying tool set is not separated from the representation during assessment. Specifically, the assessment asks the question "what is the nature of sacrifice or gain" by moving to high-level representation was addressed.

5.1 Conclusions

As documented in Chapter 4, Handel-C does not support interface representation or generation. This requires manual implementation and limits the automated exploration of partitions between hardware and software. For this work, a single interface was designed to be used by all experiments providing hardware communication and buffering and address based reading and writing for the software testbench. This experience emphasized the difficulty of custom design across the various software and hardware systems present on the

development platform. In this case it was assisted by automation software for device driver development. In an entirely custom designed system, developing the interface is likely to be the most difficult aspect of development. This is the major downfall of Handel-C as a truly unified design representation and automated interface design remains an open problem. Without this aspect, Handel-C does not fulfill all of the requirements for design representation in Hardware Software Co-design.

The experimentation undertaken for this work investigated the differences in hardware development using Handel-C versus VHDL. The software aspect of the representation is a direct ANSI-C to Handel-C translation and relies on the same current compiler technology. This focuses the assessment on why a developer would choose one design representation over the other when targeting hardware. Based on the experimental results, it is plausible to state that the final system prototypes have similar attributes (functionality, resource usage, clock speed) regardless of the design representation chosen. Considering the end product of the process is the same, the question becomes: "Is there anything in the process itself that benefits the developer?" This question appears to have an answer which depends on the context in which it is asked, specifically, during which part of the development cycle are the tools and representation being used. Table 5.1 summarizes the experimental observations which point towards the answers. A check mark in the table indicates the preferred language-tool set option. The requirements based on design level (i.e. prototype, functional, timing) favour one option over another but this is particular to the design context. For example, if simulation of functional requirements is the goal, then working with Handel-C is the preferred option. However, for a timing accurate probing of design components during design optimization, VHDL becomes the preference.

Handel-C excels in the translation from specification to implementation when a C language specification is used. An algorithm such as TEA, which can be coded in a few lines of C targeting software, can be coded in a very similar few lines of Handel-C targeting hardware. The same algorithm coded in VHDL requires far more effort and code. VHDL needs the instantiation of hardware components, design of a data path, and development of

Design Tool Observation	Handel-C	VHDL
Ease of Specification	√	
Design Verification	Prototype Level	Timing Level
Simulation	Functional Level	Timing Level
Interface	√	Hardware Only
Implementation	√	√

Table 5.1. *Summary Comparison of Hardware Tool Observations*

the control logic for capturing signals in registers to achieve the same results as a single variable assignment from an arithmetic expression statement in Handel-C. The volume of code required in VHDL to describe the same functionality as a single line of Handel-C increases the probability of coding errors and decreases understanding of the algorithm when examining the implementation code. As one examines the code listings provided in the appendices, this becomes clear.

The simplicity of the codings for designs developed in Handel-C stands out versus the VHDL codings. Considering the end prototype implementations are functionally equal with similar resource requirements and clock speeds, the simplest route to the end is the obvious choice. In prototype and iterative based design methodologies, the quick movement from specification to implementation is key to maintaining design deadlines and accelerates progress to the next prototyping or iteration cycle.

Additionally, a developer trained in C programming with a basic understanding of hardware design can successfully create a complex hardware system prototype from a specification using Handel-C. A more detailed understanding of data path and control system design is required for the same proficiency in VHDL, as well as language specific training.

The analogy between software development and system development with hardware components, drawn in Section 3.1, is reinforced by the experiences during experimentation. In software, if a particular function or section of code requires optimization beyond

the capabilities of the compiler, hand coding at the low-level of assembly language is performed. The same approach to hardware development using Handel-C can be employed. A system prototype developed quickly in Handel-C can be "profiled" through simulation to determine which components require optimization. If this is beyond the capabilities of the compiler, these components can then be hand coded directly into VHDL. VHDL enables access to low-level constructs, but this access is not required for all components, only those requiring optimization. This process could be enabled by the inclusion of EDA tool support for the translation of Handel-C to VHDL in a "user-friendly" form. The current tool set can translate a design into VHDL from the final compiler generated EDIF net-list file but not from the original Handel-C code. This form does not assist with further design optimization.

The goal to use abstraction to contend with increasing complexity is supported by Handel-C, a high-level design representation. This same goal can not be reached using VHDL or other low-level representations. It is defeated by the quantity of detail in its low-level implementation. Handel-C supports progress towards a true high-level unified design representation. It supports the flexibility to migrate design implementation between hardware and software using the same specification without translation.

5.2 Future Directions

At the time of writing, there have been several candidates for a unified design representation. It is anticipated that this trend will continue as design complexity increases leading to further improvements in representations and the accompanying automation tools.

Further experimentation leading to statistically significant comparisons between low-level and high-level representations is the next step to strengthen the argument for high-level languages. Using the classroom setting for such experimentation is more acceptable than the economic impact of conducting comparisons on a large scale in industry.

Future unified design representations must address the interface between hardware and

software components more directly. The interface should be treated as a third design space with the capability for abstraction, implementation options, as well as automated generation within the design tools for the entire system.

Bibliography

[1] *IEEE Standard Verilog Hardware Description Language.* New York, NY: IEEE, 2001.

[2] *1076 IEEE Standard VHDL Language Reference Manual.* New York, NY: IEEE, 2002.

[3] "DK Design Suite," Celoxica Limited, Abingdon, Oxfordshire, UK, Data Sheet, 2003.

[4] "DK2 Handel-C Language," Celoxica Limited, Abingdon, Oxfordshire, UK, Reference Manual, 2003.

[5] "Apex 20KC Programmable Logic Device," Altera Corporation, San Jose, California, Data Sheet Version 2.1, April 2002.

[6] "Apex PCI Development Board," Altera Corporation, San Jose, California, Data Sheet Version 2.1, April 2002.

[7] Altera Corporation, San Jose, California, PCI MegaCore Function User Guide. Version 2.0.0, August 2001.

[8] "Quartus II PLD Design," Altera Corporation, San Jose, California, Tech. Rep., December 2002. [Online]. Available: http://www.altera.com/products/software/pld/design/pld/qts-design _ flow _ pld.html

[9] "Quartus II Verification Methods," Altera Corporation, San Jose, California, Tech. Rep., December 2002. [Online]. Available: http://www.altera.com/products/software/pld/design/verification/qts-design _ flow-verification2.html♯hardware

[10] "Virtex-II Pro X Platform FPGAs: Introduction and Overview," Xilinx Inc., San Jose, California, Data Sheet Version 1.0, November 2003.

[11] "Stratix Device Family Data Sheet," Altera Corporation, San Jose, California, Data Sheet Version 2.1, October 2003.

[12] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays.* New York, New York: ACM Press. [Online]. Available: http://www.andraka.com/files/crdcsrvy.pdf

[13] W. Bishop, "Configurable computing for mainstream software applications," Ph.D. dissertation, University of Waterloo, 2003.

[14] S. Brown, R. Francis, J. Rose, and Z. Vranesic, *Field-Programmable Gate Arrays*. Boston: Kluwer Academic Publishers, 1992.

[15] S. Brown and J. Rose, "FPGA and CPLD architectures: a tutorial," *IEEE Design and Test of Computers*, vol. 13, no. 2, Summer 1996.

[16] Celoxica, "Welcome to celoxica," June 2003. [Online]. Available: http://www.celoxica.com/

[17] S. H. Chasen, "A little history of C4," in *The CAD/CAM Handbook*, C. Machover, Ed. New York: McGraw-Hill, 1996.

[18] M. Daniels, "IC design," in *The CAD/CAM Handbook*, C. Machover, Ed. New York: McGraw-Hill, 1996.

[19] R. Drechsler and B. Becker, *Binary Decision Diagrams: Theory and Implementation*. Dordrecht, Netherlands: Kluwer Academic Publishers, 1998.

[20] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1994.

[21] A. Gerstlauer, R. Dömer, J. Peng, and D. Gajski, *System Design: A Practical Guide with SpecC*. Boston: Kluwer Academic Publishers, 2001.

[22] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Norwell, Massachusetts: Kluwer Academic Publishers, 2002.

[23] R. Gupta and S. Shukla, "Panel: Should the space of implementation possibilities be determined by the abilities of high-level synthesis and validation?" in *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03), 24-26 June 2003*. IEEE Computer Society.

[24] J. Hamblen, "FPGAs for rapid prototyping," in *The Computer Engineering Handbook*, V. Oklobdzija, Ed. Boca Raton, Florida: CRC Press, 2002.

[25] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, June 1987.

[26] R. Hartenstein, "Trends in reconfigurable logic and reconfigurable computing," in *9th International Conference on Electronics, Circuits and Systems, 2002*, vol. 2, 2002.

[27] J. Henkel, X. S. Hu, and S. S. Bhattacharyya, "Taking on the embedded system design challenge," *Computer*, vol. 36, no. 4, April 2003.

[28] O. S. Initiative, "Systemc : Welcome," June 2003. [Online]. Available: http://www.systemc.org/

[29] Jungo, *WinDriver Version 6.x User's Guide.* Israel: Jungo Ltd., 2003.

[30] T. Katayama, K. Saisho, and A. Fukuda, "Proposal of a support system for device driver generation," in *Software Engineering Conference, 1999. (APSEC '99) Proceedings. Sixth Asia Pacific.* IEEE.

[31] J. Morris, "Reconfigurable computing," in *The Computer Engineering Handbook*, V. Oklobdzija, Ed. Boca Raton, Florida: CRC Press, 2002.

[32] M. Mylona, D. Holding, and K. Blow, "DES developed in Handel-C," in *Proceedings of the London Communications Symposium, 9–10 September 2002.* Dept. of E.E. Eng., University College London, Torrington Place, London: Communications Engineering Doctorate Centre. [Online]. Available: http://www.ee.acl.uk/lcs/papers2002/LCS057.pdf

[33] S. Narayan, F. Vahid, and D. Gajski, "System specification with the SpecCharts language," *Design and Test of Computers, IEEE*, vol. 9, no. 4, Dec. 1992.

[34] M. Nelson and J.-L. Gailly, *The Data Compression Book*, 2nd ed. New York, New York: M&T Books, 1996.

[35] S. T. O. C. Office, "Specc technology open consortium," June 2003. [Online]. Available: http://www.specc.gr.jp/eng/index.htm

[36] V. Oklobdzija, Ed., *The Computer Engineering Handbook.* Boca Raton, Florida: CRC Press, 2002.

[37] D. Rich, "The evolution of systemverilog," *Design and Test of Computers, IEEE*, vol. 20, no. 4, July-Aug. 2003.

[38] M. Serra and W. Gardner, "Hardware/software codesign - introducing an interdisciplinary course," in *WCCCE Conference - Vancouver, 1998.*

[39] SystemVerilog, "Systemverilog," March 2004. [Online]. Available: http://www.systemverilog.org/home.html

[40] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction.* New York: John Wiley and Sons, Inc., 2002.

[41] D. Wheeler and R. Needham, "TEA, a tiny encryption algorithm." [Online]. Available: http://www.ftp.cl.cam.ac.uk/ftp/papers/djw-rmn/djw-rmn-tea.html

[42] W. Wolf, *Computers as Components: Principles of Embedded Computing System Design.* San Francisco: Morgan Kaufmann Publishers, 2001.

[43] W. Wolf, "Embedded systems-on-chips," in *The Computer Engineering Handbook*, V. Oklobdzija, Ed. Boca Raton, Florida: CRC Press, 2002.

Appendix A

Hardware Interface Controller: VHDL Code Listing

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY ControlBlock IS
    port
    (
        clock :  IN  STD_LOGIC;
        lt_framen :  IN  STD_LOGIC;
        lt_dxfrn :  IN  STD_LOGIC;
        lt_ackn :  IN  STD_LOGIC;
        resultreq :  IN  STD_LOGIC;
        inputreq :  IN  STD_LOGIC;
        l_cmdo :  IN  STD_LOGIC_VECTOR(3 downto 0);
        l_dato :  IN  STD_LOGIC_VECTOR(31 downto 0);
        lt_tsr :  IN  STD_LOGIC_VECTOR(11 downto 0);
        result :  IN  STD_LOGIC_VECTOR(31 downto 0);

-- For Simulation purposes

--      here : inout std_logic_vector (3 downto 0);
--      read : out std_logic;
--      write : out std_logic;
--       seto : out STD_LOGIC;
--       geto : out STD_LOGIC;
--       seti : out STD_LOGIC;
--       geti : out STD_LOGIC;

-- End of Simulation pin

        lt_rdyn :  OUT  STD_LOGIC;
        lt_discn : OUT STD_LOGIC;
        lt_abortn : OUT STD_LOGIC;
        lirqn :  OUT  STD_LOGIC;
        resultrdy :  INOUT  STD_LOGIC;
        resultrec :  OUT  STD_LOGIC;
        inputrdy :  INOUT  STD_LOGIC;
        inputrec :  OUT  STD_LOGIC;
        designreset : OUT STD_LOGIC;
        input :  OUT  STD_LOGIC_VECTOR(31 downto 0);
```

```
        l_adi :  OUT  STD_LOGIC_VECTOR(31 downto 0)
    );
END ControlBlock;

ARCHITECTURE bdf_type OF ControlBlock IS

component bufferblock
    PORT(setinput :  IN  STD_LOGIC;
         getinput :  IN  STD_LOGIC;
         clock :  IN  STD_LOGIC;
         setoutput :  IN  STD_LOGIC;
         getoutput :  IN  STD_LOGIC;
         outflush :  IN  STD_LOGIC;
         allflush :  IN  STD_LOGIC;
         inflush :  IN  STD_LOGIC;
         fromdesign :  IN  STD_LOGIC_VECTOR(31 downto 0);
         fromPCI :  IN  STD_LOGIC_VECTOR(31 downto 0);
         infull :  OUT  STD_LOGIC;
         inempty :  OUT  STD_LOGIC;
         outfull :  OUT  STD_LOGIC;
         outempty :  OUT  STD_LOGIC;
         todesign :  OUT  STD_LOGIC_VECTOR(31 downto 0);
         toPCI :  OUT  STD_LOGIC_VECTOR(31 downto 0)
    );
end component;

signal PCIwrite, PCIread : STD_LOGIC;
signal setinput, getinput : STD_LOGIC;
signal setoutput, getoutput : STD_LOGIC;
signal infull, inempty : STD_LOGIC;
signal outfull, outempty : STD_LOGIC;
signal allflush, inflush, outflush : STD_LOGIC;
signal state : STD_LOGIC_VECTOR (3 DOWNTO 0);

BEGIN

b2v_inst : bufferblock
PORT MAP(setinput => setinput,
         getinput => getinput,
         clock => clock,
         setoutput => setoutput,
         getoutput => getoutput,
```

```
        outflush => outflush,
        allflush => allflush,
        inflush => inflush,
        fromdesign => result,
        fromPCI => l_dato,
        infull => infull,
        inempty => inempty,
        outfull => outfull,
        outempty => outempty,
        todesign => input,
        toPCI => l_adi);

PCIwrite <= not( l_cmdo(0) and l_cmdo(1) and l_cmdo(2) and (not l_cmdo(3)) );
PCIread <= not( (not l_cmdo(0)) and l_cmdo(1) and l_cmdo(2) and (not l_cmdo(3)) );

-- Simulation pin connections

-- here <= state;
-- write <= PCIwrite;
-- read <= PCIread;
-- geti <= getinput;
-- seti <= setinput;
-- geto <= getoutput;
-- seto <= setoutput;

-- End of simulation connnections

Machine : Process (clock) is

BEGIN

    if (clock = '1') then

    case state is

        when "0000" =>  -- Initialize the system

            lt_abortn <= '1';
            lt_discn <= '1';
            lt_rdyn <= '1';
            resultrdy <= '1';
            resultrec <= '0';
```

```
        inputrdy <= '0';
        inputrec <= '0';
        lirqn <= '1';
        allflush <= '1';
        designreset <= '1';
        getinput <= '0';
        setoutput <= '0';
        setinput <= '0';
        getoutput <= '0';

        state <= "0110";  -- Wait State

    when "0001" =>  -- System Ready State

        if ( PCIwrite or lt_framen ) = '0' then

            if ( infull or lt_tsr(1) ) = '1' then

                state <= "0110";  -- Wait State
                lt_abortn <= '0';  -- Target Abort

            else

                state <= "0011";  -- PCI Writing I State
                lt_rdyn <= '0';

            end if;

        elsif ( PCIread or lt_framen ) = '0' then

            if ( outempty or lt_tsr(0) or lt_tsr(2) ) = '1' then

                state <= "0110";  -- Wait State
                lt_abortn <= '0';  -- Target Abort

            elsif (lt_tsr(9) = '0') then

                state <= "1001"; -- PCI Single Reading I State
                lt_rdyn <= '0';

            elsif (lt_tsr(9) = '1') then
```

```
                state <= "0111"; -- PCI Burst Reading I State
                lt_rdyn <= '0';

            end if;

        elsif ( inputrdy and inputreq ) = '1' then

            state <= "0101";  -- Design Data Request State

        elsif ( resultrdy and resultreq ) = '1' then

            state <= "1101";  -- Design Result Request State

        end if;

    when "0011" =>  -- PCI Writing I State

        if ( (not lt_tsr(0)) or lt_dxfrn ) = '0' then

            setinput <= '1';
            state <= "0010";

        elsif (lt_tsr(2) and (not lt_dxfrn) ) = '1' then

            case l_dato(1 downto 0) is

                when "11" =>

                    allflush <= '1';
                    designreset <= '1';
                    state <= "0110"; -- wait state

                when "10" =>

                    inflush <= '1';
                    state <= "0110"; -- wait state

                when "01" =>

                    outflush <= '1';
                    state <= "0110"; -- wait state
```

```
            when "00" =>

                setoutput <= '1';
                state <= "0110"; -- wait state

            when OTHERS => state <= "0110"; -- wait state

        end case;

    end if;

when "0010" =>  -- PCI Writing II State

    if ( (not lt_dxfrn) and infull) = '1' then

        state <= "0110";  -- Wait State
        setinput <= '0';
        lt_rdyn <= '1';
        lt_discn <= '0';  -- Target Disconnect

    elsif lt_dxfrn = '1' then

        state <= "0110";  -- Wait State
        setinput <= '0';
        lt_rdyn <= '1';  -- Normal Completion

    end if;

when "1001" => -- PCI Single Reading I State

    if lt_ackn = '0' then

        getoutput <= '1';
        state <= "1011";  -- PCI Single Reading II State

    end if;

when "1011" =>  -- PCI Single Reading II State

    getoutput <= '0';
    if lt_dxfrn = '0' then
```

```
            state <= "1010";  -- PCI Single Reading III State

        end if;

    when "1010" =>  -- PCI Single Reading III State

        if lt_dxfrn = '1' then

            state <= "0110";  -- Wait State
            getoutput <= '0';
            lt_rdyn <= '1';  -- Normal Completion

        end if;

    when "0111" => -- PCI Burst Reading I State

        if lt_ackn = '0' then

            getoutput <= '1';
            state <= "0100";  -- PCI Burst Reading II State

        end if;

    when "0100" =>  -- PCI Burst Reading II State

        getoutput <= '0';
        if lt_dxfrn = '0' then

            getoutput <= '1';
            state <= "1100";  -- PCI Burst Reading III State

        end if;

    when "1100" =>  -- PCI Burst Reading III State

        if ( (not lt_dxfrn) and outempty) = '1' then

            state <= "0110";  -- Wait State
            getoutput <= '0';
            lt_rdyn <= '1';
            lt_discn <= '0';  -- Target Disconnect
```

```
    elsif lt_dxfrn = '1' then

        state <= "0110";  -- Wait State
        getoutput <= '0';
        lt_rdyn <= '1';  -- Normal Completion

    end if;

when "0101" =>  -- Design Data Request State

        getinput <= '1';
        inputrec <= '1';
        state <= "1110";  -- Wait Input Handshake Completion

when "1110" =>  -- Wait Input Handshake Completion

        getinput <= '0';
        if (inputreq = '0') then

            inputrec <= '0';
            state <= "0110";  -- Wait State

        end if;

when "1101" =>  -- Design Result Request State

        setoutput <= '1';
        resultrec <= '1';
        state <= "1000";  -- Wait Result Handshake Completion

when "1000" =>  -- Wait Handshake Completion

        setoutput <= '0';
        if (resultreq = '0') then

            resultrec <= '0';
            state <= "0110";  -- Wait State

        end if;

when "0110" =>  -- Wait State
```

```
            lt_abortn <= '1';
            lt_discn <= '1';
            lt_rdyn <= '1';
            allflush <= '0';
            inflush <= '0';
            outflush <= '0';
            designreset <= '0';
            getinput <= '0';
            setoutput <= '0';
            setinput <= '0';
            getoutput <= '0';

            state <= "0001";  -- Return to Ready State

        when OTHERS => state <= "0000";

    end case;

    lirqn <= '1';
    inputrdy <= not inempty;
    resultrdy <= not outfull;

    end if;

end process;

END;
```

Appendix B

Tiny Encryption Algorithm: C Code Listing

```
/*
    Software implementation and testbench for Tiny Encryption
    Algorithm. Algorithm code is used as published by D. Wheeler
    and R. Needham.
*/

#include "averager_test_lib.h"
#include "d:/windriver/samples/shared/pci_diag_lib.h"
#include <stdio.h>
#include "d:/windriver/include/status_strings.h"

// Global variables

static char line[256];
DWORD result;
AVERAGER_TEST_HANDLE hAVERAGER_TEST = NULL;
HANDLE hWD;
DWORD dwAction = 0;
BOOL fRegisteredEvent = FALSE;

/*
    Hardware location and access function provided by WinDriver.
    Returns a handle for communications with the hardware
    implementation.
*/

AVERAGER_TEST_HANDLE AVERAGER_TEST_LocateAndOpenBoard(DWORD dwVendorID, DWORD dwDeviceID)
{
    DWORD cards, my_card;
    AVERAGER_TEST_HANDLE hAVERAGER_TEST = NULL;

    if (dwVendorID==0)
    {
        printf ("Enter VendorID: ");
        fgets(line, sizeof(line), stdin);
        sscanf (line, "%x",&dwVendorID);
        if (dwVendorID==0) return NULL;

        printf ("Enter DeviceID: ");
        fgets(line, sizeof(line), stdin);
        sscanf (line, "%x",&dwDeviceID);
    }
```

```
    cards = AVERAGER_TEST_CountCards (dwVendorID, dwDeviceID);
    if (cards==0)
    {
        printf ("%s", AVERAGER_TEST_ErrorString);
        return NULL;
    }
    else if (cards==1)
        my_card = 1;
    else
    {
        UINT i;

        printf ("Found %u matching PCI cards\n", (UINT)cards);
        printf ("Select card (1-%u): ", (UINT)cards);
        i = 0;
        fgets(line, sizeof(line), stdin);
        sscanf (line, "%d",&i);
        if (i>=1 && i <=cards) my_card = i;
        else
        {
            printf ("Choice is out of range\n");
            return NULL;
        }
    }
    if (!AVERAGER_TEST_Open (&hAVERAGER_TEST, dwVendorID, dwDeviceID, my_card - 1))
    {
        printf ("%s", AVERAGER_TEST_ErrorString);
        return NULL;
    }
    printf ("PCI card found!\n");
    return hAVERAGER_TEST;
}

/*
    Function code takes two 32-bit words as the array
    v and the encoding 128-bit key k as parameters.
    The software version is commented out and the
    hardware communications are active.

    In software, the words v[0] and v[1] are encoded using
    the TEA and the key k. The encoded values are
    copied back to the array in place of the original values.
```

```
    In hardware, the all clear signal is sent followed by the
    two words for encoding. The encoded words are retrieved by
    polling and copied back to the original array. The key
    has been implemented in hardware.
*/

void code (DWORD* v, DWORD* k) {
    //DWORD y = v[0], z = v[1], sum = 0, delta = 0x9e3779b9, n = 32;

    int count = 3;
    AVERAGER_TEST_WriteDword(hAVERAGER_TEST, 2, 0, 3);

    AVERAGER_TEST_WriteDword(hAVERAGER_TEST, 0, 0, v[0]);
    AVERAGER_TEST_WriteDword(hAVERAGER_TEST, 0, 0, v[1]);

/*
    Software specification for TEA, removed by commenting.
    Software interface substituted to allow communication
    with hardware implementations.


    while (n -- > 0) {
      sum += delta;
      y += (z<<4)+k[0] ^ z+sum ^ (z>>5) + k[1] ;
      z += (y<<4)+k[2] ^ y+sum ^ (y>>5) + k[3] ;
    }
    v[0] = y;
    v[1] = z;

*/

    while (count > 0) {

        do {
            result = AVERAGER_TEST_ReadDword(hAVERAGER_TEST, 1, 0);
        } while (result == 0xffffffff);

        switch (count) {

            case 3 : count --;
                break;
```

```
            case 2 : v[0] = result;
                count --;
                AVERAGER_TEST_WriteDword(hAVERAGER_TEST, 2, 0, 0);
                break;
            case 1 : v[1] = result;
                count --;
                break;
            default : break;
        }
    }
}

/*
    Function decode takes the encoded values and the 128-bit
    key and copies the decoded values back to the original
    array.
*/

void decode (DWORD* v, DWORD* k) {
    DWORD y = v[0], z = v[1], sum, delta = 0x9e3779b9, n = 32;

    sum = delta<<5;

    while (n -- > 0) {

        z -= (y<<4)+k[2] ^ y+sum ^ (y>>5) + k[3] ;
        y -= (z<<4)+k[0] ^ z+sum ^ (z>>5) + k[1] ;
        sum -= delta;
    }
    v[0] = y;
    v[1] = z;
}

/*
    Testbench function: sends three simple word parings to the
    code function, displays the results, decodes the encodings, and
    displays the results.
*/

int main(int argc, char *argv[])
{
    DWORD * vee;
```

```
DWORD * key;

if (!PCI_Get_WD_handle(&hWD))
    return 0;
WD_Close (hWD);

hAVERAGER_TEST = AVERAGER_TEST_LocateAndOpenBoard(AVERAGER_TEST_DEFAULT_VENDOR_ID, AVERAGER_TEST_DEFA

key = (DWORD *) malloc(4*(sizeof(DWORD)));
key[0] = 0x11112222;
key[1] = 0x33334444;
key[2] = 0x55556666;
key[3] = 0x77778888;

vee = (DWORD *) malloc(2*sizeof(DWORD));

vee[0] = 0x12345678;
vee[1] = 0x87654321;

code (vee, key);
printf("Code: %x, %x ",vee[0],vee[1]);

decode (vee, key);
printf("Decode: %x, %x\n",vee[0],vee[1]);

vee[0] = 0x11223344;
vee[1] = 0x55667788;

code (vee, key);
printf("Code: %x, %x ",vee[0],vee[1]);

decode (vee, key);
printf("Decode: %x, %x\n",vee[0],vee[1]);

vee[0] = 0x12128c8c;
vee[1] = 0x66996699;

code (vee, key);
printf("Code: %x, %x ",vee[0],vee[1]);

decode (vee, key);
printf("Decode: %x, %x\n",vee[0],vee[1]);
```

```
    free(key);
    free(vee);

}
```

Appendix C

Tiny Encryption Algorithm: Handel-C Code Listing

```
// Set target technology

set family = AlteraApex20KC;
set part = "EP20K1000CF672C7";

// Define in ports for hardware interface

interface port_in(unsigned 1 clk) ClockPort();
interface port_in(unsigned 1 resultrdy) ResultRdy();
interface port_in(unsigned 1 resultrec) ResultRec();
interface port_in(unsigned 1 inputrdy) InputRdy();
interface port_in(unsigned 1 inputrec) InputRec();
interface port_in(unsigned 1 flush) Flush();
interface port_in(unsigned int 32 input) Input() with {std_logic_vector = 0};

// Set clock and asynchronous reset

set clock = internal ClockPort.clk;
set reset = internal Flush.flush;

// Declare internal signals

unsigned 1 resrq;
unsigned 1 inprq;
unsigned int 32 a;
unsigned int 32 b;
unsigned int 32 c;

unsigned int 32 k0;
unsigned int 32 k1;
unsigned int 32 k2;
unsigned int 32 k3;
unsigned int 32 d;

unsigned int 32 sum;
unsigned int 6 n;

// Define out ports for hardware interface

interface port_out() ResultReq(unsigned 1 resultreq = resrq);
interface port_out() InputReq(unsigned 1 inputreq = inprq);
```

```
interface port_out() Result(unsigned int 32 result = c) with {std_logic_vector = 0};


void main(void) {

// Set hardware definitions for encryption key and delta

    k0 = 0x11112222;
    k1 = 0x33334444;
    k2 = 0x55556666;
    k3 = 0x77778888;
    d = 0x9e3779b9;

    while ( 1 ) {

        n = 32;

// Begin data read protocol with hardware interface

        while  (InputRdy.inputrdy == 0) {
            delay;
        }

        inprq = 1;

        while (InputRec.inputrec == 0) {
            delay;
        }

        par {
            a = Input.input;
            inprq = 0;
        }

        delay;

        while (InputRdy.inputrdy == 0) {
            delay;
        }

        inprq = 1;
```

```
        while (InputRec.inputrec == 0) {
            delay;
        }

        par {
            b = Input.input;
            inprq = 0;
        }

        delay;

// End data read protocol with hardware interface

// Encode the data words

        while (n > 0) {

            seq {
                sum += d;
                a += (b << 4) + k0 ^ b + sum ^ (b >> 5) + k1;
                b += (a << 4) + k2 ^ a + sum ^ (a >> 5) + k3;
                n = n - 1;
            }

        }

// Begin result write protocol with hardware interface

        par {
            c = a;
            resrq = 1;
        }

        while (ResultRec.resultrec == 0) {
            delay;
        }

        resrq = 0;
        delay;

        par {
            c = b;
```

```
            resrq = 1;
        }

        while (ResultRec.resultrec == 0) {
            delay;
        }

        resrq = 0;

// End result write protocol with hardware interface

    }
}
```

Appendix D

Tiny Encryption Algorithm: VHDL Code Listing

```
-- VHDL implementation of TEA. Implements
-- the hardware interface for inclusion in testbench.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

-- Implement hardware interface

ENTITY TinyExp1 IS
    PORT(
         clk : IN STD_LOGIC;
         resultrdy : IN STD_LOGIC;
         resultrec : IN STD_LOGIC;
         inputrdy : IN STD_LOGIC;
         inputrec : IN STD_LOGIC;
         flush : IN STD_LOGIC;
         input : IN STD_LOGIC_VECTOR(31 downto 0);
         resultreq : OUT STD_LOGIC;
         inputreq : OUT STD_LOGIC;
         result : OUT STD_LOGIC_VECTOR(31 downto 0)
    );
END TinyExp1;

-- Architecture uses structural components built
-- with parametric function provided in the Quartus II
-- environment. Components reside in common directory
-- with project design files.

ARCHITECTURE bdf_type OF TinyExp1 IS

-- 32-bitwise exclusive or

component lpm_xor0
    PORT(data0x : IN STD_LOGIC_VECTOR(31 downto 0);
         data1x : IN STD_LOGIC_VECTOR(31 downto 0);
         result : OUT STD_LOGIC_VECTOR(31 downto 0)
    );

end component;
```

```
-- 32-bit register with enable and asynchronous clear

component lpm_dff3
    PORT(clock :  IN STD_LOGIC;
         enable :  IN STD_LOGIC;
         aclr : IN STD_LOGIC;
         data :  IN STD_LOGIC_VECTOR(31 downto 0);
         q : OUT STD_LOGIC_VECTOR(31 downto 0)
    );

end component;

-- 8-bit register with enable and asynchronous clear

component lpm_dff4
    PORT(clock :  IN STD_LOGIC;
         enable :  IN STD_LOGIC;
         aclr : IN STD_LOGIC;
         data :  IN STD_LOGIC_VECTOR(7 downto 0);
         q : OUT STD_LOGIC_VECTOR(7 downto 0)
    );

end component;

-- 32-bit adder

component lpm_add_sub1
    PORT(dataa : IN STD_LOGIC_VECTOR(31 downto 0);
         datab : IN STD_LOGIC_VECTOR(31 downto 0);
         result : OUT STD_LOGIC_VECTOR(31 downto 0)
    );

end component;

-- 8-bit incrementer

component lpm_add_sub2
    PORT(dataa : IN STD_LOGIC_VECTOR(7 downto 0);
         result : OUT STD_LOGIC_VECTOR(7 downto 0)
    );
```

```
end component;

signal  state :  STD_LOGIC_VECTOR(3 downto 0);
signal  count :  STD_LOGIC_VECTOR(7 downto 0);
signal  Delta :  STD_LOGIC_VECTOR(31 downto 0);
signal  encount :  STD_LOGIC;
signal  ensum :  STD_LOGIC;
signal  eny :  STD_LOGIC;
signal  enz :  STD_LOGIC;
signal  incount :  STD_LOGIC_VECTOR(7 downto 0);
signal  insum :  STD_LOGIC_VECTOR(31 downto 0);
signal  iny :  STD_LOGIC_VECTOR(31 downto 0);
signal  inz :  STD_LOGIC_VECTOR(31 downto 0);
signal  tempy :  STD_LOGIC_VECTOR(31 downto 0);
signal  tempz :  STD_LOGIC_VECTOR(31 downto 0);
signal  k0 :  STD_LOGIC_VECTOR(31 downto 0);
signal  k1 :  STD_LOGIC_VECTOR(31 downto 0);
signal  k2 :  STD_LOGIC_VECTOR(31 downto 0);
signal  k3 :  STD_LOGIC_VECTOR(31 downto 0);
signal  sum :  STD_LOGIC_VECTOR(31 downto 0);
signal  y :  STD_LOGIC_VECTOR(31 downto 0);
signal  z :  STD_LOGIC_VECTOR(31 downto 0);
signal  SYNTHESIZED_WIRE_0 :  STD_LOGIC_VECTOR(31 downto 0);
signal  SYNTHESIZED_WIRE_1 :  STD_LOGIC_VECTOR(31 downto 0);
signal  SYNTHESIZED_WIRE_2 :  STD_LOGIC_VECTOR(31 downto 0);
signal  SYNTHESIZED_WIRE_3 :  STD_LOGIC_VECTOR(31 downto 0);
signal  SYNTHESIZED_WIRE_4 :  STD_LOGIC_VECTOR(31 downto 0);
signal  SYNTHESIZED_WIRE_5 :  STD_LOGIC_VECTOR(31 downto 0);
signal  SYNTHESIZED_WIRE_6 :  STD_LOGIC_VECTOR(31 downto 0);
signal  SYNTHESIZED_WIRE_7 :  STD_LOGIC_VECTOR(31 downto 0);
signal  SYNTHESIZED_WIRE_8 :  STD_LOGIC_VECTOR(31 downto 0);
signal  SYNTHESIZED_WIRE_9 :  STD_LOGIC_VECTOR(31 downto 0);
signal  y4 :  STD_LOGIC_VECTOR(31 downto 0);
signal  y5 :  STD_LOGIC_VECTOR(31 downto 0);
signal  z4 :  STD_LOGIC_VECTOR(31 downto 0);
signal  z5 :  STD_LOGIC_VECTOR(31 downto 0);

BEGIN

-- dataflow layout of components

y4(31 downto 4) <= y(27 downto 0);
```

```
y4(3 downto 0) <= "0000";
y5(26 downto 0) <= y(31 downto 5);
y5(31 downto 27) <= "00000";

z4(31 downto 4) <= z(27 downto 0);
z4(3 downto 0) <= "0000";
z5(26 downto 0) <= z(31 downto 5);
z5(31 downto 27) <= "00000";

Delta <= "10011110001101110111100110111001";
k0 <= "00010001000100010010001000100010";
k1 <= "00110011001100110100010001000100";
k2 <= "01010101010101010110011001100110";
k3 <= "01110111011101111000100010001000";


insty : lpm_dff3
PORT MAP(data => iny,
         clock => clk,
         enable => eny,
         aclr => flush,
         q => y);

instz : lpm_dff3
PORT MAP(data => inz,
         clock => clk,
         enable => enz,
         aclr => flush,
         q => z);

instsum : lpm_dff3
PORT MAP(data => insum,
         clock => clk,
         enable => ensum,
         aclr => flush,
         q => sum);

Counter : lpm_dff4
PORT MAP(data => incount,
         clock => clk,
         enable => encount,
         aclr => flush,
```

```
        q => count);

inst : lpm_add_sub1
PORT MAP(dataa => sum,
         datab => Delta,
         result => insum);

inst4 : lpm_add_sub1
PORT MAP(dataa => z,
         datab => sum,
         result => SYNTHESIZED_WIRE_0);

inst5 : lpm_add_sub1
PORT MAP(dataa => z4,
         datab => k0,
         result => SYNTHESIZED_WIRE_1);

inst6 : lpm_add_sub1
PORT MAP(dataa => z5,
         datab => k1,
         result => SYNTHESIZED_WIRE_3);

inst7 : lpm_add_sub1
PORT MAP(dataa => y,
         datab => sum,
         result => SYNTHESIZED_WIRE_4);

inst8 : lpm_add_sub1
PORT MAP(dataa => y4,
         datab => k2,
         result => SYNTHESIZED_WIRE_5);

inst9 : lpm_add_sub1
PORT MAP(dataa => y5,
         datab => k3,
         result => SYNTHESIZED_WIRE_7);

inst10 : lpm_add_sub2
PORT MAP(dataa => count,
         result => incount);

b2v_inst14 : lpm_xor0
```

```
PORT MAP(data0x => SYNTHESIZED_WIRE_0,
         data1x => SYNTHESIZED_WIRE_1,
         result => SYNTHESIZED_WIRE_2);

b2v_inst15 : lpm_xor0
PORT MAP(data0x => SYNTHESIZED_WIRE_2,
         data1x => SYNTHESIZED_WIRE_3,
         result => SYNTHESIZED_WIRE_8);

b2v_inst16 : lpm_xor0
PORT MAP(data0x => SYNTHESIZED_WIRE_4,
         data1x => SYNTHESIZED_WIRE_5,
         result => SYNTHESIZED_WIRE_6);

b2v_inst17 : lpm_xor0
PORT MAP(data0x => SYNTHESIZED_WIRE_6,
         data1x => SYNTHESIZED_WIRE_7,
         result => SYNTHESIZED_WIRE_9);

inst11 : lpm_add_sub1
PORT MAP(dataa => SYNTHESIZED_WIRE_8,
         datab => y,
         result => tempy);

inst12 : lpm_add_sub1
PORT MAP(dataa => SYNTHESIZED_WIRE_9,
         datab => z,
         result => tempz);

-- control flow finite state machine

fsm : process (clk) is

BEGIN

    if (clk = '1') then

    case state is

        when "0000" => -- ready/empty state

            if inputrdy = '1' then
```

```
            state <= "0001";
            inputreq <= '1';

        end if;

    when "0001" => -- get y input

        if inputrec = '1' then

            iny <= input;
            state <= "0011";
            eny <= '1';
            inputreq <= '0';

        end if;

    when "0011" => -- ready for z

        eny <= '0';

        if inputrdy = '1' then

            state <= "1011";
            inputreq <= '1';

        end if;

    when "1011" => -- get z input

        if inputrec = '1' then

            inz <= input;
            state <= "0010";
            enz <= '1';
            inputreq <= '0';

        end if;

    when "0010" => --  Begin Processing

        enz <= '0';
```

```
        encount <= '1';


        if count(7) = '1' then

            state <= "0110"; -- Send Results
            encount <= '0';
            ensum <= '0';

        elsif count (1 downto 0) = "01" then

            state <= "1010"; -- Ready y
            encount <= '0';
            ensum <= '1';

        end if;

    when "1010" => -- Ready y

        ensum <= '0';
        state <= "1000"; -- Update y

    when "1000" => -- Update y

        iny <= tempy;
        encount <= '1';
        eny <= '1';

        if count(1 downto 0) = "10" then

            state <= "1100"; -- Ready z
            encount <= '0';

        end if;

    when "1100" => -- Ready z

        eny <= '0';
        state <= "1101"; -- Update z

    when "1101" => -- Update z
```

```
        inz <= tempz;
        encount <= '1';
        enz <= '1';

        if count (1 downto 0) = "11" then

            enz <= '0';
            state <= "1001"; -- End of loop

        end if;

    when "1001" => -- End of loop

        enz <= '0';
        state <= "0010"; -- Begin Processing

    when "0110" => -- request send y results

        if resultrdy = '1' then

            state <= "0100";
            result <= y;
            resultreq <= '1';

        end if;

    when "0100" => -- send y results

        if resultrec = '1' then

            state <= "0101";
            resultreq <= '0';

        end if;

    when "0101" => -- request send z results

        if resultrdy = '1' then

            state <= "0111";
            result <= z;
            resultreq <= '1';
```

```
            end if;

        when "0111" => -- send z results

            if resultrec = '1' then

                state <= "0000";
                resultreq <= '0';

            end if;

        when OTHERS =>

            state <= "0000";

        end case;

    end if;

end process fsm;


END;
```

Appendix E

JPEG: C Code Listing

```
/* This file provides a template for JPEG (or other
   compression) of greyscale images.
   Author:  R. Nigel Horspool, June 2003
*/
#include "jpgcompress.h"
#include "bmp.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "hardEncode.h"

/*  mycos[u][x] holds the precomputed values of cos( (2x+1).u.pi/8 )  */
static float mycos[4][4];

/*  c[u]  contains the individual scaling coefficients for DCT terms  */
static float c[4];

/*  scale is the overall scaling factor for DCT calculations  */
static float scale;


/* quantization table for 4 x 4 matrix example */
static int Q[4][4] = {
    {16,10,24,51},
    {14,16,40,69},
    {18,37,68,103},
    {49,78,103,120}
};

/* Previous F(0,0) value for encoding */
static int FPrev;

/* zigzag pattern for encoding */
struct zigzag {
    int row;
    int col;
} ZigZag [16] = {
    {0,0},
    {0,1}, {1,0},
    {2,0}, {1,1}, {0,2},
    {0,3}, {1,2}, {2,1}, {3,0},
```

```
    {3,1}, {2,2}, {1,3},
    {2,3}, {3,2},
    {3,3}
    };



/*  init() initializes the pre-computed constants  */
static void init() {
    const double pi = 3.14159265358979;
    int x, u;

    for( u = 0;  u < 4;  u++ ) {
    for( x = 0;  x < 4;  x++ ) {
        mycos[u][x] = (float) cos((2.0*x+1.0)*u*pi/8.0);

    }
    }
    scale = c[0] = (float) (sqrt(2.0) * 0.5);
    c[1] = c[2] = c[3] = 1.0;
    FPrev = 0;
}


/* fdct2 takes a 4 X 4 matrix and performs a 2-D DCT
   by applying 1-D DCT to the rows and then applying
   1-D DCT to the columns.
*/
void fdct2( float arr[4][4] ) {
    int row, col;
    float sum, temp[4];
    int x, u;

    /* apply 1-D DCT to the rows */
    for( row = 0;  row < 4;  row++ ) {
        for( u = 0;  u < 4;  u++ ) {
            sum = 0.0;
            for( x = 0;  x < 4;  x++ )
                sum += arr[row][x] * mycos[u][x];
            temp[u] = sum*c[u]*scale;
        }
        for( u = 0;  u < 4;  u++ )
```

```
            arr[row][u] = temp[u];
    }

    /* apply 1-D DCT to the columns */
    for( col = 0;  col < 4;  col++ ) {
        for( u = 0;  u < 4;  u++ ) {
            sum = 0.0;
            for( x = 0;  x < 4;  x++ )
                sum += arr[x][col] * mycos[u][x];
            temp[u] = sum*c[u]*scale;
            }
        for( u = 0;  u < 4;  u++ )
            arr[u][col] = temp[u];
    }

}

/* edct2 takes a 4 X 4 matrix and performs an inverse 2-D DCT
   by applying 1-D inverse DCT to the columns and then applying
   1-D inverse DCT to the rows.
*/
void idct2( float arr[4][4] ) {
    int row, col;
    float sum, temp[4];
    int x, u;

    /* apply 1-D inverse DCT to the cols */
    for( col = 0;  col < 4;  col++ ) {
        for( x = 0;  x < 4;  x++ ) {
        sum = 0.0;
        for( u = 0;  u < 4;  u++ )
            sum += c[u] * arr[u][col] * mycos[u][x];
            temp[x] = sum*scale;
        }
        for( x = 0;  x < 4;  x++ )
        arr[x][col] = temp[x];
    }

    /* apply 1-D inverse DCT to the rows */
    for( row = 0;  row < 4;  row++ ) {
        for( x = 0;  x < 4;  x++ ) {
        sum = 0.0;
```

```
        for( u = 0;  u < 4;  u++ )
            sum += c[u] * arr[row][u] * mycos[u][x];
            temp[x] = sum*scale;
        }
        for( x = 0;  x < 4;  x++ ) {
            arr[row][x] = (float)(int)temp[x];
        }
    }
}

/* quantize and round the matrix */
void quant(float arr[4][4]) {
    int row, col;

    for ( row = 0; row < 4; row++) {
    for ( col = 0; col < 4; col++) {

        arr[row][col] = (float)(int)(arr[row][col]/Q[row][col]);

    }
    }

}


/* de (quantize and round) the matrix */
void dequant(float arr[4][4]) {
    int row, col;

    for ( row = 0; row < 4; row++) {
    for ( col = 0; col < 4; col++) {

        arr[row][col] = arr[row][col]*Q[row][col];

    }
    }
}


/* LenAmp writes the length and amplitude values to
   the output file.
*/
```

```
void LenAmp (int code, int zeros, FILE *ofile) {
    int Len = 1;
    int neg = 0;

    if (code < 0) {
        neg = 1;
        code = 0 - code;
    }

    if (zeros == 0) {

        if (code == 0) {

            fputc (0, ofile);
            fputc (0, ofile);

        } else {

            while ( code > (int)((pow(2,Len)) - 1) ) {
                Len ++;
            }

            fputc (Len, ofile);

            if (neg == 1) {
                fputc ( ~(code - (int)(pow(2,(Len -1)))), ofile);
            } else {
                fputc (code - (int)(pow(2,(Len -1))), ofile);
            }
        }
    } else {

        fputc (code,ofile);

    }
}

/*  outputEncode encodes the DCT coefficients for output
    and sends the information to LenAmp for writing to the
    output file.
*/
void outputEncode(float arr[4][4], FILE *ofile) {
```

```
    int TempF;
    int i;
    int zeroCount;
    int code;

    TempF = (int) arr[0][0];
    arr[0][0] = FPrev - arr[0][0];
    FPrev = TempF;

    LenAmp ((int)arr[0][0],0,ofile);

    zeroCount = 0;

    for(i = 1; i < 16; i++) {
        code = (int)arr[ZigZag[i].row][ZigZag[i].col];
        if (code == 0)
            zeroCount ++;
        else {
            LenAmp (zeroCount, 1, ofile);
            zeroCount = 0;
            LenAmp (code, 0, ofile);
        }
    }
    if (zeroCount != 0)
    LenAmp (zeroCount, 1, ofile);
}

/*  inputRead reads in the encoded length amplitude
    information in the compressed file and re-generates
    a DCT coefficient matrix.
*/
void inputRead (float arr[4][4], FILE *infile) {
    int count = 0;
    int Len, Amp, code;
    int wordlimit = 0;
    int i;

    while (count < 16) {


        if (count == 0) {
            Len = fgetc (infile);
```

```
        Amp = fgetc (infile);
        if (Amp > 127) {
            Amp = ~Amp;
            code = Amp + (int)(pow(2,(Len - 1)));
            code = 0 - code;
        } else {
            code = Amp + (int)(pow(2,(Len - 1)));
        }
        code = FPrev - code;
        FPrev = code;
        arr[0][0] = (float) code;
        count ++;
        wordlimit ++;

    } else {

        if (wordlimit == 0) {

            Len = fgetc (infile);
            Amp = fgetc (infile);
            if (Amp > 127) {
                Amp = ~Amp;
                code = Amp + (int)(pow(2,(Len - 1)));
                code = 0 - code;
            } else {
                code = Amp + (int)(pow(2,(Len - 1)));
            }
            arr[ZigZag[count].row][ZigZag[count].col] = (float) code;
            count ++;
            wordlimit ++;

        } else {
            Len = fgetc (infile);
            wordlimit = 0;
            if (Len != 0) {
                for (i = 0; i < Len; i++) {
                    if (count > 15) break;
                    arr[ZigZag[count].row][ZigZag[count].col] = 0;
                    count ++;
                }
            }
        }
```

```
        }
    }
}

/* Compress an image using JPEG compression.
   Input parameters are:
     ofilename = the name of the compressed file to create
     rowPtr = an array of pointers to row data; rowPtr is
              indexed by the row number, and the result is
              indexed by the column number.
              The result is an unsigned byte containing a
              greyscale value from 0 to 255 for that pixel.
     width = image width in pixels
     height = image heigh in pixels.
     debug = flag to enable debugging output.
   The function result is
     0 to indicate that a compressed file was successfully created,
     non-zero to indicate an error.
*/
int jpgcompress( char *ofilename, BYTE **rowPtr, int width, int height,
    int debug ) {
    FILE *ofile;
    int row, col, smallRow, smallCol;

    float val[4][4];

    init();

    ofile = fopen(ofilename, "wb");
    if (ofile == NULL) {
        perror(ofilename);
        return 1;
    }

    if (debug) {
    fprintf(stderr, "Parameters: width=%d, height=%d\n", width, height);
    }

    /* output file signature in first 2 bytes */
    writeInt16(ofile, JPGSIGNATURE);

    /* output width as 2 bytes */
```

```
    writeInt16(ofile, width);

    /* output height as 2 bytes */
    writeInt16(ofile, height);

    /* now compress the greyscale image
       using 4 X 4 matrices */
    for( row = 0;  row < height;  row = row + 4 ) {
    for( col = 0;  col < width;  col = col + 4 ) {

        for (smallRow = 0; smallRow < 4; smallRow ++) {
        for (smallCol = 0; smallCol < 4; smallCol ++) {
            val[smallRow][smallCol] = (float) (rowPtr[row + smallRow][col + smallCol] - 128.0);
        }
        }

        //fdct2 (val);  // software DCT transform of 4 x 4 matrix

        hardEncode (val); // hardware DCT transform of 4 x 4 matrix

        quant (val);  // quantize and round transformed matrix

        outputEncode (val, ofile); //write the encoding to file

    }
    }

    fclose(ofile);
    return 0;
}


/*  jpgdecompress -- reads a compressed image from the file named
    by the first argument.
    Input parameters:
      filename = the name of the file to read a compressed image from,
      debug    = flag to enable debugging output.

    Output parameters:
      rowPtrPtr = address where a pointer to a table of row pointers
                  is to be stored.
                  This function creates the table of row pointers,
```

```
                where indexing by row number gives row data vector;
                indexing row data by a column number gives the
                greyscale intensity for the pixel at row,col
      widthPtr = address where the image width must be stored
      heightPtr = address where the image height must be stored.

    A non-zero function result indicates failure.
*/
int jpgdecompress( char *filename, BYTE ***rowPtrPtr,
            int *widthPtr, int *heightPtr, int debug ) {
    FILE *infile;
    int width, height, bytesPerRow;
    BYTE **rowPtr, *p, *bits;
    int i;
    int signature;
    int row, col, smallRow, smallCol;

    float val[4][4];

    init();

    infile = fopen(filename, "rb");
    if (infile == NULL) {
    perror(filename);
    return 1;
    }

    /* read and check the two signature bytes */
    signature = readInt16(infile);
    if (signature != JPGSIGNATURE) {
    fprintf(stderr, "File %s does not begin with correct signature\n",
        filename);
    return 1;
    }

    /* read the image width and height */
    width = readInt16(infile);
    height = readInt16(infile);

    if (debug) {
    fprintf(stderr, "Read signature=%d, width = %d, height = %d\n",
        signature, width, height);
```

```
}

/* allocate memory for the array of row pointers */
rowPtr = calloc(height, sizeof(BYTE*));

/* allocate memory for all the rows of the image    */
/* (all bits in the memory are initialized to zero) */
bytesPerRow = (width+3) & (~3);
p = bits = calloc(height,bytesPerRow);
if (rowPtr == NULL || bits == NULL) {
fprintf(stderr, "Memory allocation failed\n");
return 1;
}

/* initialize the array of row pointers */
for( i = 0;  i < height;  i++ ) {
rowPtr[i] = bits;
bits += bytesPerRow;
}

/* decompress each row of the input */
for( row = 0;  row < height;  row = row + 4 ) {
for( col = 0;  col < width;  col = col + 4 ) {
    inputRead (val, infile);

    dequant (val);  // quantize and round transformed matrix
    idct2 (val);  // dct transform of 4 x 4 matrix

    for (smallRow = 0; smallRow < 4; smallRow ++) {
    for (smallCol = 0; smallCol < 4; smallCol ++) {
        rowPtr[row + smallRow][col + smallCol] = (int)val[smallRow][smallCol] + 128;
    }
    }
}
}

/* return the width and height to the caller */
*widthPtr = width;
*heightPtr = height;

/* return the row pointer array to the caller */
*rowPtrPtr = rowPtr;
```

```
    return 0;
}
```

```
/*
    Software testbench for DCT operation as part of JPEG Algorithm.
*/

#include <stdio.h>
#include <math.h>
#include <time.h>
#include "d:/windriver/wizard/my_projects/averager_test_lib.h"
#include "d:/windriver/samples/shared/pci_diag_lib.h"
#include "d:/windriver/include/status_strings.h"

// Global variables

static char line[256];
DWORD result;
AVERAGER_TEST_HANDLE hAVERAGER_TEST = NULL;
HANDLE hWD;
DWORD dwAction = 0;
BOOL fRegisteredEvent = FALSE;


/*
    Hardware location and access function provided by WinDriver.
    Returns a handle for communications with the hardware
    implementation.
*/

AVERAGER_TEST_HANDLE AVERAGER_TEST_LocateAndOpenBoard(DWORD dwVendorID, DWORD dwDeviceID)
{
    DWORD cards, my_card;
    AVERAGER_TEST_HANDLE hAVERAGER_TEST = NULL;

    if (dwVendorID==0)
    {
        printf ("Enter VendorID: ");
        fgets(line, sizeof(line), stdin);
        sscanf (line, "%x",&dwVendorID);
        if (dwVendorID==0) return NULL;

        printf ("Enter DeviceID: ");
        fgets(line, sizeof(line), stdin);
```

```
        sscanf (line, "%x",&dwDeviceID);
    }
    cards = AVERAGER_TEST_CountCards (dwVendorID, dwDeviceID);
    if (cards==0)
    {
        printf ("%s", AVERAGER_TEST_ErrorString);
        return NULL;
    }
    else if (cards==1)
        my_card = 1;
    else
    {
        UINT i;

        printf ("Found %u matching PCI cards\n", (UINT)cards);
        printf ("Select card (1-%u): ", (UINT)cards);
        i = 0;
        fgets(line, sizeof(line), stdin);
        sscanf (line, "%d",&i);
        if (i>=1 && i <=cards) my_card = i;
        else
        {
            printf ("Choice is out of range\n");
            return NULL;
        }
    }
    if (!AVERAGER_TEST_Open (&hAVERAGER_TEST, dwVendorID, dwDeviceID, my_card - 1))
    {
        printf ("%s", AVERAGER_TEST_ErrorString);
        return NULL;
    }
    printf ("PCI card found!\n");
    return hAVERAGER_TEST;
}

/*
    Function hardEncode takes s 4 X 4 matrix and send the members to
    hardware in row order. Results are polled and replace the original
    members of the matrix.
*/
int hardEncode (float arr[4][4]) {
```

```
int count = 0;
int hole = 0;
time_t start, end;
DWORD data;

if (!PCI_Get_WD_handle(&hWD))
    return 0;
WD_Close (hWD);

if (hAVERAGER_TEST == NULL) {

hAVERAGER_TEST = AVERAGER_TEST_LocateAndOpenBoard(AVERAGER_TEST_DEFAULT_VENDOR_ID, AVERAGER_TEST_DEFA

}
AVERAGER_TEST_WriteDword(hAVERAGER_TEST, 2, 0, 3);

while (count < 16) {

    data = (DWORD) (arr[count/4][count%4] * 256);
    AVERAGER_TEST_WriteDword(hAVERAGER_TEST, 0, 0, data);
    count ++;

}
count = 0;

while (count < 16) {

    time(&start);
    do {
        result = AVERAGER_TEST_ReadDword(hAVERAGER_TEST, 1, 0);
        time(&end);
    } while ((result == 0xffffffff) && (difftime(end,start) < 1)) ;

    if (hole == 0) {
        hole ++;

    } else {
        arr[count/4][count%4] =(float) (signed int)result;
        count ++;
    }
}
```

```
    return 1;

}
```

Appendix F

JPEG: Handel-C Code Listing

```
// Set target technology

set family = AlteraApex20KC;
set part = "EP20K1000CF672C7";

// Define in ports for hardware interface

interface port_in(unsigned 1 clk) ClockPort();
interface port_in(unsigned 1 resultrdy) ResultRdy();
interface port_in(unsigned 1 resultrec) ResultRec();
interface port_in(unsigned 1 inputrdy) InputRdy();
interface port_in(unsigned 1 inputrec) InputRec();
interface port_in(unsigned 1 flush) Flush();
interface port_in(unsigned int 32 input) Input() with {std_logic_vector = 0};

// Set clock and asynchronous reset
// Clock divided to meet timing requirements

set clock = internal_divide ClockPort.clk 10;
set reset = internal Flush.flush;

// Declare internal signals

unsigned 1 resrq;
unsigned 1 inprq;
ram signed int 32 a[4][4];
unsigned int 32 b;

unsigned int 2 row, col;
signed int 32 sum;
signed int 32 sumtemp;
signed int 32 midtemp;
signed int 32 transtemp1;
signed int 32 transtemp2;
signed int 32 temp[4];
unsigned int 2 x, u;


// Define out ports for hardware interface

interface port_out() ResultReq(unsigned 1 resultreq = resrq);
```

```
interface port_out() InputReq(unsigned 1 inputreq = inprq);
interface port_out() Result(unsigned int 32 result = b) with {std_logic_vector = 0};


signed int 32 scale = 0x0000000b5;

// Define ROM's for lookup values

static rom signed int 32 c [4] = {
        0x000000b5,
        0x00000100,
        0x00000100,
        0x00000100
};

static rom signed int 32 mycos [4][4] = {
        {0x00000100, 0x00000100, 0x00000100, 0x00000100},
        {0x000000ec, 0x00000062, 0xffffff9e, 0xffffff14},
        {0x000000b5, 0xffffff4b, 0xffffff4b, 0x000000b5},
        {0x00000062, 0xffffff14, 0x000000ec, 0xffffff9e}
};


// Round takes a signed 32-bit input and returns the value
// shifted right by 8 bits and rounded based on bit 7.

signed int 32 Round (signed int 32 num) {

    if (num[7] == 1)

        return ((num >> 8) + 1);

    else

        return (num >>8);

}

// Performs DCT on rows of a 4 X 4 matrix
// based on 8-bit fixed point percision.

void DCT (void) {
```

```
row = 0;
do {

    u = 0;
    do {

        par {
            sum = 0;
            x = 0;
        }
        do {

            seq {
                sumtemp = a[row][x] * mycos[u][x];
                sumtemp = Round(sumtemp);
                sum = sum + sumtemp;
                x ++;
            }

        } while ( x > 0);

        seq {
            midtemp = sum*c[u];
            midtemp = Round(midtemp);
            midtemp = midtemp*scale;
            temp[u] = Round(midtemp);
            u ++;
        }


    } while (u > 0);

    u = 0;
    do {

        seq {
            a[row][u] = temp[u];
            u ++;
        }

    } while (u > 0);
```

```
        row ++;
        } while (row > 0);

}

// Transposes a 4 X 4 matrix

void Transpose(void) {

        row = 0;
        do {

            col = row + 1;
            do {

                seq {

                    transtemp1 = a [row][col];
                    transtemp2 = a [col][row];
                    a [row][col] = transtemp2;
                    a [col][row] = transtemp1;
                }

            col ++;
            } while (col > 0);

        row ++;
        } while (row < 3);

}

void main(void) {

    while ( 1 ) {

// Begin data read protocol with hardware interface
// for 16 input values and store in RAM structure.

        row = 0;
        do {
```

```
            col = 0;
            do {

                while  (InputRdy.inputrdy == 0) {
                    delay;
                }

                inprq = 1;

                while (InputRec.inputrec == 0) {
                    delay;
                }

                par {
                    a [row][col] = (signed) Input.input;
                    inprq = 0;
                }

                delay;

            col ++;
            } while (col > 0);

        row ++;
        } while (row > 0);

// End data read protocol with hardware interface


// Perform 2-D DCT by employing row based 1-D DCT and
// transposing the matrix.

        DCT();
        Transpose();
        DCT();
        Transpose();



// Begin result write protocol with hardware interface

       row = 0;
```

```
        do {

            col = 0;
            do {

                par {
                    b = (unsigned) Round(a[row][col]);
                    resrq = 1;
                }

                while (ResultRec.resultrec == 0) {
                    delay;
                }

                resrq = 0;
                delay;

            col ++;
            } while (col > 0);

        row ++;
        } while (row > 0);

// End result write protocol with hardware interface

    }
}
```

Appendix G

JPEG: VHDL Code Listing

```
-- VHDL implementation of DCT required for 4 X 4 matrix
-- manipulation as part of JPEG compression. Implements
-- the hardware interface for inclusion in testbench.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

-- Implement hardware interface

ENTITY JPEGTEMP IS
    port
    (
        resultrdy :  IN  STD_LOGIC;
        resultrec :  IN  STD_LOGIC;
        inputrdy :  IN  STD_LOGIC;
        inputrec :  IN  STD_LOGIC;
        flush :  IN  STD_LOGIC;
        clk :  IN  STD_LOGIC;
        input :  IN  STD_LOGIC_VECTOR(31 downto 0);
        resultreq :  OUT  STD_LOGIC;
        inputreq :  OUT  STD_LOGIC;
        result :  OUT  STD_LOGIC_VECTOR(31 downto 0)
    );
END JPEGTEMP;

-- Architecture uses structural components built
-- with parametric function provided in the Quartus II
-- environment. Components reside in common directory
-- with project design files.

ARCHITECTURE bdf_type OF JPEGTEMP IS

-- 5-bit counter with enable and asynchronous clear

component counter_1 IS
    PORT
    (
        clock       : IN STD_LOGIC ;
        cnt_en      : IN STD_LOGIC ;
```

```
        aclr        : IN STD_LOGIC ;
        q       : OUT STD_LOGIC_VECTOR (4 DOWNTO 0)
    );
END component;

-- 2-bit counter with enable and asynchronous clear

component count_2 IS
    PORT
    (
        clock       : IN STD_LOGIC ;
        cnt_en      : IN STD_LOGIC ;
        aclr        : IN STD_LOGIC ;
        q       : OUT STD_LOGIC_VECTOR (1 DOWNTO 0)
    );
END component;

-- 4-bit counter with enable and asynchronous clear

component count_6 IS
    PORT
    (
        clock       : IN STD_LOGIC ;
        cnt_en      : IN STD_LOGIC ;
        aclr        : IN STD_LOGIC ;
        q       : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
    );
END component;

-- 7-bit register with asynchronous clear

component Reg_7 IS
    PORT
    (
        clock       : IN STD_LOGIC ;
        aclr        : IN STD_LOGIC ;
        data        : IN STD_LOGIC_VECTOR (6 DOWNTO 0);
        q       : OUT STD_LOGIC_VECTOR (6 DOWNTO 0)
    );
END component;

-- Asynchronous RAM structure 16 X 32-bit words
```

```
component lpm_ram_dp0 IS
    PORT
    (
        data        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        wraddress       : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
        rdaddress       : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
        wren        : IN STD_LOGIC;
        q       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END component;

-- Asynchronous RAM structure 4 X 32-bit words

component Ram4_32 IS
    PORT
    (
        data        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        wraddress       : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
        rdaddress       : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
        wren        : IN STD_LOGIC  := '1';
        q       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END component;

-- Multiplier: Assumes 8-bit fixed point, truncates high bits

component roundmult IS
    port
    (
        dataa       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        datab       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        result      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END component;

-- Rounds signed 8-bit fixed point to signed integer

component finalround IS
    port
    (
        dataa       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
```

```
        result      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END component;

-- ROM: 16 X 32-bit initialized to DCT cosine values

component mycos IS
    PORT
    (
        address     : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
        inclock     : IN STD_LOGIC ;
        outclock    : IN STD_LOGIC ;
        q           : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END component;

-- ROM: 4 X 32-bit initialized to DCT C array values

component c IS
    PORT
    (
        address     : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
        inclock     : IN STD_LOGIC ;
        outclock    : IN STD_LOGIC ;
        q           : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END component;

-- 32-bit adder

component lpm_add_sub0 IS
    PORT
    (
        dataa       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        datab       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        result      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END component;

-- 32-bit register with enable and asynchronous clear

component lpm_dff3 IS
```

```
    PORT
    (
        clock       : IN STD_LOGIC ;
        enable      : IN STD_LOGIC ;
        aclr        : IN STD_LOGIC ;
        data        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        q       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END component;

-- 4-bit register with enable and asynchronous clear

component addreg IS
    PORT
    (
        clock       : IN STD_LOGIC ;
        enable      : IN STD_LOGIC ;
        aclr        : IN STD_LOGIC ;
        data        : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
        q       : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
    );
END component;

-- 2-bit register with enable and asynchronous clear

component addreg2 IS
    PORT
    (
        clock       : IN STD_LOGIC ;
        enable      : IN STD_LOGIC ;
        aclr        : IN STD_LOGIC ;
        data        : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
        q       : OUT STD_LOGIC_VECTOR (1 DOWNTO 0)
    );
END component;

signal  state, nextstate : STD_LOGIC_vector (6 downto 0);
signal  scale :  STD_LOGIC_vector (31 downto 0);
signal  loopon, loopclr, asynloop : STD_LOGIC;
signal  loopcount, nextloop : STD_LOGIC_vector (4 downto 0);
signal  ensum, zerosum, asynsum : STD_LOGIC;
signal  storea, indata, dctdata : STD_LOGIC_VECTOR (31 DOWNTO 0);
```

```
signal  endata, clrdata : STD_LOGIC;
signal  rdaddress, wraddress : STD_LOGIC_VECTOR (3 DOWNTO 0);
signal  readreg, writereg : STD_LOGIC_VECTOR (3 DOWNTO 0);
signal  dctread, dctwrite : STD_LOGIC_VECTOR (3 DOWNTO 0);
signal  writeon : STD_LOGIC;
signal  enwrad, enrdad : std_logic;
signal  arr : STD_LOGIC_VECTOR (31 DOWNTO 0);
signal  sum, insum, presum, tempsum : STD_LOGIC_VECTOR (31 DOWNTO 0);
signal  DCTindex : STD_LOGIc_Vector (3 downto 0);
signal  Cindex : STD_LOGIC_vector (1 downto 0);
signal  DCTValue, CValue : STD_LOGIC_vector (31 downto 0);
signal  temp1  :  STD_LOGIC_vector (31 downto 0);
signal  temp1in :  STD_LOGIC_vector (31 downto 0);
signal  entemp1, entempsum :  Std_logic;
signal  clrinner, clrout, clrc : std_logic;
signal  incinner, incout, incc : std_logic;
signal  inner : std_logic_vector (3 downto 0);
signal  outer, cloop : std_logic_vector (1 downto 0);
signal  storetemp, intempdata, temp : std_logic_vector (31 downto 0);
signal  wrtemp, rdtemp, tempwrad, temprdad : std_logic_vector (1 downto 0);
signal  tempon, entemp, entempwr, entemprd : std_logic;
signal  outputround :  STD_LOGIC_vector (31 downto 0);


BEGIN

-- dataflow layout of components

inoutloop : counter_1
PORT map(
        clock => clk,
        cnt_en => loopon,
        aclr => asynloop,
        q => loopcount);

innerprocessloop : count_6
PORT map(
        clock => clk,
        cnt_en => incinner,
        aclr => clrinner,
        q => inner);
```

```
outprocessloop : count_2
PORT map(
        clock => clk,
        cnt_en => incout,
        aclr => clrout,
        q => outer);

cprocessloop : count_2
PORT map(
        clock => clk,
        cnt_en => incc,
        aclr => clrc,
        q => cloop);

statereg : Reg_7
PORT Map(
        clock => not(clk),
        aclr => flush,
        data => nextstate,
        q => state);

multipler1 : roundmult
Port Map(dataa => arr,
        datab => DCTValue,
        result => tempsum);

multipler2 : roundmult
Port Map(dataa => sum,
        datab => CValue,
        result => temp1in);

multipler3 : roundmult
Port Map(dataa => temp1,
        datab => scale,
        result => intempdata);

rounder : finalround
Port Map(dataa => temp,
        result => outputround);

arrstore : lpm_ram_dp0
PORT map(data => storea,
```

```
        wraddress => wraddress,
        rdaddress => rdaddress,
        wren => writeon,
        q => arr);

datain : lpm_dff3
PORT map(
        clock => clk,
        enable => endata,
        aclr => flush,
        data => indata,
        q => storea);

writeadd : addreg
PORT map(
        clock => clk,
        enable => enwrad,
        aclr => flush,
        data => writereg,
        q => wraddress);

readadd : addreg
PORT map(
        clock => clk,
        enable => enrdad,
        aclr => flush,
        data => readreg,
        q => rdaddress);

tempstore : Ram4_32
PORT map(
        data => storetemp,
        wraddress => wrtemp,
        rdaddress => rdtemp,
        wren => tempon,
        q => temp);


tempdata : lpm_dff3
PORT map(
        clock => clk,
        enable => entemp,
```

```
        aclr => flush,
        data => intempdata,
        q => storetemp);

tempwradd : addreg2
PORT map(
        clock => clk,
        enable => entempwr,
        aclr => flush,
        data => tempwrad,
        q => wrtemp);

temprdadd : addreg2
PORT map(
        clock => clk,
        enable => entemprd,
        aclr => flush,
        data => temprdad,
        q => rdtemp);

temp1store : lpm_dff3
PORT map(
        clock => clk,
        enable => entemp1,
        aclr => flush,
        data => temp1in,
        q => temp1);

sumstore : lpm_dff3
PORT map(
        clock => clk,
        enable => ensum,
        aclr => asynsum,
        data => insum,
        q => sum);

summer : lpm_add_sub0
PORT Map(
        dataa => sum,
        datab => tempsum,
        result => insum);
```

```
DCT : mycos
PORT Map(
        address => DCTindex (3 downto 0),
        inclock => clk,
        outclock => clk,
        q => DCTValue (31 downto 0));

Cvector : c
PORT Map
    (
        address => Cindex (1 downto 0),
        inclock => clk,
        outclock => clk,
        q => CValue (31 downto 0));

scale <= "00000000000000000000000010110101";
asynloop <= flush or loopclr;
asynsum <= flush or zerosum;
result <= arr;


-- control flow finite state machine

fsm : process (clk) is

BEGIN

    if (clk = '1') then

        case state is

        when "0000000" => -- Reset/Initialization State

            loopon <= '0';
            loopclr <= '1';
            writeon <= '0';
            endata <= '0';
            enwrad <= '0';
            enrdad <= '0';
            nextstate <= "0000001";

        when "0000001" =>  -- Begin data input loop
```

```
        loopclr <= '0';

        if inputrdy = '1' then

            nextstate <= "0000010";

        end if;

    when "0000010" => -- Aquire data when ready

        writereg <= loopcount (3 downto 0);
        inputreq <= '1';
        enwrad <= '1';

        if inputrec = '1' then

            nextstate <= "0000011";

        end if;

    when "0000011" => -- Begin store data in write register

        enwrad <= '0';
        indata <= input;
        endata <= '1';
        nextstate <= "0000100";

    when "0000100" => -- End store data in write register

        endata <= '0';
        inputreq <= '0';
        nextstate <= "0000101";

    when "0000101" => -- Increment loopcounter and store in RAM

        loopon <= '1';
        writeon <= '1';
        nextstate <= "0000110";

    when "0000110" => -- End store in RAM
```

```
        loopon <= '0';
        writeon <= '0';
        nextstate <= "0000111";

    when "0000111" => -- Check loopcounter for completion

        if loopcount(4) = '1' then

            nextstate <= "0001000";

        else

            nextstate <= "0000010";

        end if;

    when "0001000" => -- Start DCT processing of data

        nextstate <= "1100000"; -- Start DCT

    when "0001001" => -- Reset loopcounter for output results

        loopclr <= '1';
        nextstate <= "0001010";

    when "0001010" => -- Begin output result loop

        loopclr <= '0';
        enrdad <= '1';
        readreg <= loopcount(3 downto 0);

        if resultrdy = '1' then

            nextstate <= "0001011";

        end if;

    when "0001011" => -- Check results received

        enrdad <= '0';
        resultreq <= '1';
```

```
            if resultrec = '1' then

                nextstate <= "0001100";

            end if;

        when "0001100" => -- Increment loopcounter

            resultreq <= '0';
            loopon <= '1';
            nextstate <= "0001101";

        when "0001101" => -- Check for end of result loop

            loopon <= '0';

            if loopcount(4) = '1' then

                nextstate <= "0000000";

            else

                nextstate <= "0001010";

            end if;

-- Begin the DCT processing of input data

-- row DCT

        when "1100000" => -- Initialize processing structure

            ensum <= '0';
            entemp1 <= '0';
            entemp <= '0';
            entempwr <= '0';
            entemprd <= '0';
            clrinner <= '1';
            clrout <= '1';
            clrc <= '1';
            nextstate <= "1100001";
```

```
when "1100001" => -- Begin outer loop for DCT

    zerosum <= '1';
    clrinner <= '0';
    clrout <= '0';
    clrc <= '0';
    nextstate <= "1100010";

when "1100010" => -- Begin inner loop for DCT, set addresses

    zerosum <= '0';
    readreg <= outer & inner(1 downto 0);
    enrdad <= '1';
    DCTindex <= inner;
    nextstate <= "1100011";

when "1100011" =>

    enrdad <= '0';
    nextstate <= "1100100";

when "1100100" => -- Set additional addresses

    Cindex <= inner(3 downto 2);
    entempwr <= '1';
    tempwrad <= inner(3 downto 2);
    nextstate <= "1100101";

when "1100101" =>

    entempwr <= '0';
    nextstate <= "1100110";

when "1100110" => -- Increment inner loop

    incinner <= '1';
    nextstate <= "1100111";

when "1100111" => -- Capture sum value

    incinner <= '0';
    ensum <= '1';
```

```
        nextstate <= "1101000";

    when "1101000" => -- Check for end of inner loop

        ensum <= '0';

        if inner(1 downto 0) = "00" then

            nextstate <= "1101001";

        else

            nextstate <= "1100010";

        end if;

    when "1101001" => -- Allow sufficient clock for multiplication

        null;
        nextstate <= "1101010";

    when "1101010" =>

        null;
        nextstate <= "1101011";

    when "1101011" =>

        entemp1 <= '1';
        nextstate <= "1101100";

    when "1101100" =>

        entemp1 <= '0';
        nextstate <= "1101101";

    when "1101101" => -- Allow sufficient clock for multiplication

        null;
        nextstate <= "1101110";

    when "1101110" =>
```

```
        null;
        nextstate <= "1101111";

    when "1101111" =>

        entemp <= '1';
        nextstate <= "1110000";

    when "1110000"=>

        entemp <= '0';
        nextstate <= "1110001";

    when "1110001" => -- Store temp value for update of data

        tempon <= '1';
        nextstate <= "1110010";

    when "1110010" => -- Check for end of column loop

        tempon <= '0';

        if inner(3 downto 2) = "00" then

            nextstate <= "1110011";

        else

            nextstate <= "1100001";

        end if;

    when "1110011" =>

        clrc <= '1';
        nextstate <= "1110100";

    when "1110100" => -- Begin loop to transfer temp to data

        clrc <= '0';
        temprdad <= cloop;
```

```
        entemprd <= '1';
        nextstate <= "1110101";

    when "1110101" =>

        entemprd <= '0';
        nextstate <= "1110110";

    when "1110110" => -- Set write address

        writereg <= outer & cloop;
        enwrad <= '1';
        indata <= temp;
        endata <= '1';
        nextstate <= "1110111";

    when "1110111" => -- Sequence control signals

        enwrad <= '0';
        endata <= '0';
        writeon <= '1';
        nextstate <= "1111000";

    when "1111000" => -- Increment transfer loop

        writeon <= '0';
        incc <= '1';
        nextstate <= "1111001";

    when "1111001" =>

        incc <= '0';
        nextstate <= "1111010";

    when "1111010" => -- Check for end of transfer loop

        if cloop = "00" then

            nextstate <= "1111100";

        else
```

```
                nextstate <= "1110100";

            end if;

        when "1111100" => -- Increment outer loop

            incout <= '1';
            nextstate <= "1111101";

        when "1111101" =>

            incout <= '0';
            nextstate <= "1111110";

        when "1111110" => -- Check for end of row loop

            if outer = "00" then

                nextstate <= "1111111";

            else

                nextstate <= "1100001";

            end if;

        when "1111111" =>

            nextstate <= "1000000";

-- column DCT

        when "1000000" => -- Initialize the processing structure

            ensum <= '0';
            entemp1 <= '0';
            entemp <= '0';
            entempwr <= '0';
            entemprd <= '0';
            clrinner <= '1';
            clrout <= '1';
            clrc <= '1';
```

```
        nextstate <= "1000001";

    when "1000001" => -- Begin outer column loop

        zerosum <= '1';
        clrinner <= '0';
        clrout <= '0';
        clrc <= '0';
        nextstate <= "1000010";

    when "1000010" => -- Set addresses

        zerosum <= '0';
        readreg <= inner(1 downto 0) & outer;
        enrdad <= '1';
        DCTindex <= inner;
        nextstate <= "1000011";

    when "1000011" =>

        enrdad <= '0';
        nextstate <= "1000100";

    when "1000100" => -- Set additional addresses

        Cindex <= inner(3 downto 2);
        entempwr <= '1';
        tempwrad <= inner(3 downto 2);
        nextstate <= "1000101";

    when "1000101" =>

        entempwr <= '0';
        nextstate <= "1000110";

    when "1000110" => -- Increment inner loop

        incinner <= '1';
        nextstate <= "1000111";

    when "1000111" =>
```

```
        incinner <= '0';
        ensum <= '1';
        nextstate <= "1001000";

    when "1001000" => -- Check for end of inner loop

        ensum <= '0';

        if inner(1 downto 0) = "00" then

            nextstate <= "1001001";

        else

            nextstate <= "1000010";

        end if;

    when "1001001" => -- Allow sufficient clocks for multiplication

        null;
        nextstate <= "1001010";

    when "1001010" =>

        null;
        nextstate <= "1001011";

    when "1001011" =>

        entemp1 <= '1';
        nextstate <= "1001100";

    when "1001100" =>

        entemp1 <= '0';
        nextstate <= "1001101";

    when "1001101" =>

        null;
        nextstate <= "1001110";
```

```
when "1001110" =>

    null;
    nextstate <= "1001111";

when "1001111" =>

    entemp <= '1';
    nextstate <= "1010000";

when "1010000"=>

    entemp <= '0';
    nextstate <= "1010001";

when "1010001" => -- Update temp value for transfer to data

    tempon <= '1';
    nextstate <= "1010010";

when "1010010" => -- Check for end of inner loop

    tempon <= '0';

    if inner(3 downto 2) = "00" then

        nextstate <= "1010011";

    else

        nextstate <= "1000001";

    end if;

when "1010011" => -- Initialize transfer loop

    clrc <= '1';
    nextstate <= "1010100";

when "1010100" => -- Begin transfer loop
```

```
        clrc <= '0';
        temprdad <= cloop;
        entemprd <= '1';
        nextstate <= "1010101";

    when "1010101" =>

        entemprd <= '0';
        nextstate <= "1010110";

    when "1010110" => -- Set addresses for transfer to data

        writereg <= cloop & outer;
        enwrad <= '1';
        indata <= outputround;
        endata <= '1';
        nextstate <= "1010111";

    when "1010111" =>

        enwrad <= '0';
        endata <= '0';
        writeon <= '1';
        nextstate <= "1011000";

    when "1011000" => -- Increment transfer loop

        writeon <= '0';
        incc <= '1';
        nextstate <= "1011001";

    when "1011001" =>

        incc <= '0';
        nextstate <= "1011010";

    when "1011010" => -- Check for end of transfer loop

        if cloop = "00" then

            nextstate <= "1011100";
```

```
            else

                nextstate <= "1010100";

            end if;

        when "1011100" => -- Increment column loop

            incout <= '1';
            nextstate <= "1011101";

        when "1011101" =>

            incout <= '0';
            nextstate <= "1011110";

        when "1011110" => -- Check for end of column loop

            if outer = "00" then

                nextstate <= "1011111";

            else

                nextstate <= "1000001";

            end if;

        when "1011111" => -- DCT complete, send results

            nextstate <= "0001001";

        when OTHERS =>

            nextstate <= "0000000";

        end case;

    end if;

end process fsm;
```

```
END;
```

Appendix H

Cordic: C Code Listing

```
/*
    Software implementation and testbench for CORDIC function.
    Based on functional descriptions provided by Ray Andraka in
    "A survey of CORDIC algoritms for FPGA based computers",
    FPGA 98 Monterey CA USA Copyright 1998 ACM.
*/

#include <stdio.h>
#include <math.h>
#include <time.h>
#include "d:/windriver/wizard/my_projects/averager_test_lib.h"
#include "d:/windriver/samples/shared/pci_diag_lib.h"
#include "d:/windriver/include/status_strings.h"

// Global Variables

static char line[256];
AVERAGER_TEST_HANDLE hAVERAGER_TEST = NULL;
HANDLE hWD;
DWORD dwAction = 0;
BOOL fRegisteredEvent = FALSE;


// arctan of 1/(2^-i) in degrees, lookup table for input to Cordic function

float epsilon [] = {45.000000, 26.565051, 14.036243, 7.125016, 3.576334, 1.789911, 0.895174, 0.447614,
0.223811, 0.111906, 0.055953, 0.027976, 0.013988, 0.006994, 0.003497, 0.001749};

/*
    Hardware location and access function provided by WinDriver.
    Returns a handle for communications with the hardware
    implementation.
*/
AVERAGER_TEST_HANDLE AVERAGER_TEST_LocateAndOpenBoard(DWORD dwVendorID, DWORD dwDeviceID)
{
    DWORD cards, my_card;
    AVERAGER_TEST_HANDLE hAVERAGER_TEST = NULL;

    if (dwVendorID==0)
    {
        printf ("Enter VendorID: ");
```

```
        fgets(line, sizeof(line), stdin);
        sscanf (line, "%x",&dwVendorID);
        if (dwVendorID==0) return NULL;

        printf ("Enter DeviceID: ");
        fgets(line, sizeof(line), stdin);
        sscanf (line, "%x",&dwDeviceID);
    }
    cards = AVERAGER_TEST_CountCards (dwVendorID, dwDeviceID);
    if (cards==0)
    {
        printf ("%s", AVERAGER_TEST_ErrorString);
        return NULL;
    }
    else if (cards==1)
        my_card = 1;
    else
    {
        UINT i;

        printf ("Found %u matching PCI cards\n", (UINT)cards);
        printf ("Select card (1-%u): ", (UINT)cards);
        i = 0;
        fgets(line, sizeof(line), stdin);
        sscanf (line, "%d",&i);
        if (i>=1 && i <=cards) my_card = i;
        else
        {
            printf ("Choice is out of range\n");
            return NULL;
        }
    }
    if (!AVERAGER_TEST_Open (&hAVERAGER_TEST, dwVendorID, dwDeviceID, my_card - 1))
    {
        printf ("%s", AVERAGER_TEST_ErrorString);
        return NULL;
    }
    printf ("PCI card found!\n");
    return hAVERAGER_TEST;
}

/*
```

```
    Function doCordic takes 3 values for x, y, z and
    performs calculations in rotational mode.

    In software, 16 iteration of the function are performed.

    In hardware, the all clear signal is sent followed by the
    x, y, and z values left shifted by 12 for fixed point
    calculations. The results of the rotation are returned.
*/

void doCordic (float *x, float *y, float *z) {

    int i;
    //float x_temp;

    DWORD data;
    DWORD result;
    time_t start, end;


    if (!PCI_Get_WD_handle(&hWD))
        return 0;
    WD_Close (hWD);

    if (hAVERAGER_TEST == NULL) {

    hAVERAGER_TEST = AVERAGER_TEST_LocateAndOpenBoard(AVERAGER_TEST_DEFAULT_VENDOR_ID, AVERAGER_TEST_DEFAU

    }
    AVERAGER_TEST_WriteDword(hAVERAGER_TEST, 2, 0, 3);


    data = (DWORD) ((int)(*x * 4096));
    AVERAGER_TEST_WriteDword(hAVERAGER_TEST, 0, 0, data);
    data = (DWORD) ((int)(*y * 4096));
    AVERAGER_TEST_WriteDword(hAVERAGER_TEST, 0, 0, data);
    data = (DWORD) ((int)(*z * 4096));
    AVERAGER_TEST_WriteDword(hAVERAGER_TEST, 0, 0, data);



    time(&start);
```

```
    do {
        result = AVERAGER_TEST_ReadDword(hAVERAGER_TEST, 1, 0);
        time(&end);
    } while ((result == 0xffffffff) && (difftime(end,start) < 1)) ;

    time(&start);
    do {
        result = AVERAGER_TEST_ReadDword(hAVERAGER_TEST, 1, 0);
        time(&end);
    } while ((result == 0xffffffff) && (difftime(end,start) < 1)) ;

    *x = ((float) (signed int) result) / 4096.0;

    time(&start);
    do {
        result = AVERAGER_TEST_ReadDword(hAVERAGER_TEST, 1, 0);
        time(&end);
    } while ((result == 0xffffffff) && (difftime(end,start) < 1)) ;

    *y = ((float) (signed int) result) / 4096.0;

    AVERAGER_TEST_WriteDword(hAVERAGER_TEST, 2, 0, 0);

    time(&start);
    do {
        result = AVERAGER_TEST_ReadDword(hAVERAGER_TEST, 1, 0);
        time(&end);
    } while ((result == 0xffffffff) && (difftime(end,start) < 1)) ;

    *z = ((float) (signed int) result) / 4096.0;

/*
    Software specification for Cordic, removed by commenting.
    Software interface substituted to allow communication
    with hardware implementations.

    for (i = 0; i < 16; i ++) {

      x_temp = *x;

      if (*z >= 0.0) {
```

```
            *x = *x - (*y * pow(2,-i));
            *y = *y + (x_temp * pow(2,-i));
            *z = *z - epsilon[i];

        } else {

            *x = *x + (*y  * pow(2,-i));
            *y = *y - (x_temp * pow(2,-i));
            *z = *z + epsilon[i];

        }

    }

*/

}


/*
    Testbench function: receives an angle in degrees from
    the user and returns the sine and cosine as calculated
    by the Cordic implementation.
*/

int main( int argc, char *argv[] ) {

    float x, y, z;
    int angle;

    printf("Cordic Calculator for angles between 0 and +90 degrees\n");
    do {
      angle = 0;

      do {
        printf("\nEnter angle in degrees (>90 to quit) : ");
        scanf("%i", &angle);
      } while (angle<0);

      if (angle <= 90) {
        x = 0.6073;
        y = 0.0;
```

```
        z = (float) angle;

        doCordic (&x,&y,&z);

        printf("%d degrees: sin = %.4f cos = %.4f\n",angle,y, x);

      }

    } while (angle <= 90);
    exit(0);
}
```

Appendix I

Cordic: Handel-C Code Listing

```
// Set target technology

set family = AlteraApex20KC;
set part = "EP20K1000CF672C7";

// Define in ports for hardware interface

interface port_in(unsigned 1 clk) ClockPort();
interface port_in(unsigned 1 resultrdy) ResultRdy();
interface port_in(unsigned 1 resultrec) ResultRec();
interface port_in(unsigned 1 inputrdy) InputRdy();
interface port_in(unsigned 1 inputrec) InputRec();
interface port_in(unsigned 1 flush) Flush();
interface port_in(unsigned int 32 input) Input() with {std_logic_vector = 0};

// Set clock and asynchronous reset
// Clock divided to meet timing requirements

set clock = internal_divide ClockPort.clk 5;
set reset = internal Flush.flush;

// Declare internal signals

unsigned 1 resrq;
unsigned 1 inprq;

signed int 32 x, y, z, x_temp;
unsigned int 32 b;

unsigned int 5 iteration;

// Define out ports for hardware interface

interface port_out() ResultReq(unsigned 1 resultreq = resrq);
interface port_out() InputReq(unsigned 1 inputreq = inprq);
interface port_out() Result(unsigned int 32 result = b) with {std_logic_vector = 0};

// Define ROM for lookup values

static rom signed int 32 arctan [16] = {0x0002D000, 0x0001A90A, 0x0000E094, 0x00007200,
                                        0x00003938, 0x00001CA3, 0x00000E52, 0x00000729,
```

```
                                    0x00000394, 0x000001CA, 0x000000E5, 0x00000072,
                                    0x00000039, 0x0000001C, 0x0000000E, 0x00000007};


void main(void) {

    while ( 1 ) {

// Begin data read protocol with hardware interface
// 3 times for x, y, and z

        while  (InputRdy.inputrdy == 0) {
            delay;
        }

        inprq = 1;

        while (InputRec.inputrec == 0) {
            delay;
        }

        par {
            x = (signed)Input.input;
            inprq = 0;
        }

        delay;

        while (InputRdy.inputrdy == 0) {
            delay;
        }

        inprq = 1;

        while (InputRec.inputrec == 0) {
            delay;
        }

        par {
            y = (signed)Input.input;
            inprq = 0;
        }
```

```
        delay;

        while  (InputRdy.inputrdy == 0) {
            delay;
        }

        inprq = 1;

        while (InputRec.inputrec == 0) {
            delay;
        }

        par {
            z = (signed)Input.input;
            inprq = 0;
        }

        delay;

// End data read protocol with hardware interface

// Perform Cordic rotation

        delay;

        for (iteration = 0; iteration < 16; iteration++) {

            x_temp = x;

            if ( z >=  0) {

                seq {
                    x = x - (y >> (0 @ iteration));
                    y = y + (x_temp >> (0 @ iteration));
                    z = z - arctan[iteration[3:0]];
                }

            } else {

                seq {
                    x = x + (y >> (0 @ iteration));
```

```
                y = y - (x_temp >> (0 @ iteration));
                z = z + arctan[iteration[3:0]];
            }
        }
    }

// Begin result write protocol with hardware interface
// 3 times for x, y, and z

    while(ResultRdy.resultrdy == 0) {
        delay;
    }

    par {
        b = (unsigned)x;
        resrq = 1;
    }

    while (ResultRec.resultrec == 0) {
        delay;
    }

    resrq = 0;
    delay;

    while(ResultRdy.resultrdy == 0) {
        delay;
    }

    par {
        b = (unsigned)y;
        resrq = 1;
    }

    while (ResultRec.resultrec == 0) {
        delay;
    }

    resrq = 0;
    delay;

    while(ResultRdy.resultrdy == 0) {
```

```
            delay;
        }

        par {
            b = (unsigned)z;
            resrq = 1;
        }

        while (ResultRec.resultrec == 0) {
            delay;
        }

        resrq = 0;

// End result write protocol with hardware interface

    }
}
```

Appendix J

Cordic: VHDL Code Listing

```
-- VHDL implementation of Cordic function. Implements
-- the hardware interface for inclusion in testbench.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

-- Implement hardware interface

ENTITY Cordic IS
    port
    (
        resultrdy :  IN  STD_LOGIC;
        resultrec :  IN  STD_LOGIC;
        inputrdy :  IN  STD_LOGIC;
        inputrec :  IN  STD_LOGIC;
        flush :  IN  STD_LOGIC;
        clk :  IN  STD_LOGIC;
        input :  IN  STD_LOGIC_VECTOR(31 downto 0);
        resultreq :  OUT  STD_LOGIC;
        inputreq :  OUT  STD_LOGIC;
        result :  OUT  STD_LOGIC_VECTOR(31 downto 0)
    );
END Cordic;

-- Architecture uses structural components built
-- with parametric function provided in the Quartus II
-- environment. Components reside in common directory
-- with project design files.

ARCHITECTURE simple OF Cordic IS

-- 32-bit register with enable and asynchronous clear

component lpm_dff3 IS
    PORT
    (
        clock       : IN STD_LOGIC ;
        enable      : IN STD_LOGIC ;
        aclr        : IN STD_LOGIC ;
```

```
        data        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        q       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END component;

-- 4-bit counter with enable and asynchronous clear

component count_6 IS
    PORT
    (
        clock       : IN STD_LOGIC ;
        cnt_en      : IN STD_LOGIC ;
        aclr        : IN STD_LOGIC ;
        q       : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
    );
END component;

-- 5-bit register with asynchronous clear

component statereg IS
    PORT
    (
        clock       : IN STD_LOGIC ;
        aclr        : IN STD_LOGIC ;
        data        : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        q       : OUT STD_LOGIC_VECTOR (4 DOWNTO 0)
    );
END component;

-- 32-bit adder and subtractor

component adder_subber IS
    PORT
    (
        add_sub     : IN STD_LOGIC ;
        dataa       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        datab       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        result      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END component;

-- 32-bit arithmetic right shift by input amount
```

```
component Shifty IS
    port
    (
        input :  IN  STD_LOGIC_VECTOR(31 downto 0);
        amount : in std_logic_vector(3 downto 0);
        result :  OUT  STD_LOGIC_VECTOR(31 downto 0)
    );
END component;

-- ROM: 16 X 32-bit initialized to arctan values

component arcTan IS
    PORT
    (
        address     : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
        q       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END component;

signal in_x, next_x, x, in_y, next_y, y, in_z, next_z, z : std_logic_vector (31 downto 0);
signal en_x, en_y, en_z : std_logic;
signal iteration : std_logic_vector (3 downto 0);
signal iterate : std_logic;
signal arcTanOut, sh_x, sh_y : std_logic_vector (31 downto 0);
signal state, nextstate : std_logic_vector (4 downto 0);

begin

-- dataflow layout of components

xvalue : lpm_dff3
PORT map(
        clock => clk,
        enable => en_x,
        aclr => flush,
        data => in_x,
        q => x);

yvalue : lpm_dff3
PORT map(
        clock => clk,
```

```
        enable => en_y,
        aclr => flush,
        data => in_y,
        q => y);

zvalue : lpm_dff3
PORT map(
        clock => clk,
        enable => en_z,
        aclr => flush,
        data => in_z,
        q => z);

innerprocessloop : count_6
PORT map(
        clock => clk,
        cnt_en => iterate,
        aclr => flush,
        q => iteration);

xadder : adder_subber
Port map(
        add_sub => z(31),
        dataa => x,
        datab => sh_y,
        result => next_x);

yadder : adder_subber
Port map(
        add_sub => not(z(31)),
        dataa => y,
        datab => sh_x,
        result => next_y);

zadder : adder_subber
Port map(
        add_sub => z(31),
        dataa => z,
        datab => arcTanOut,
        result => next_z);

xshift : Shifty
```

```
Port map(
        input => x,
        amount => iteration,
        result => sh_x);

yshift : Shifty
Port map(
        input => y,
        amount => iteration,
        result => sh_y);

arcTanLUT : arcTan
Port map(
        address => iteration,
        q => arcTanOut);

FSM : statereg
Port map(
        clock => not(clk),
        aclr => flush,
        data => nextstate,
        q => state);

-- control flow finite state machine

machine : process (clk) is

BEGIN

    if (clk = '1') then

        case state is

        when "00000" => -- initialize cordic function

            iterate <= '0';
            nextstate <= "00001";

        when "00001" => -- wait for input

            if inputrdy = '1' then
```

```
            nextstate <= "00010";

        end if;

    when "00010" => -- request x input

        inputreq <= '1';

        if inputrec = '1' then

            nextstate <= "00011";

        end if;

    when "00011" => -- get x input

        in_x <= input;
        en_x <= '1';
        nextstate <= "00100";

    when "00100" => -- close x input

        en_x <= '0';
        inputreq <= '0';
        nextstate <= "00101";

    when "00101" => -- wait for input

        if inputrdy = '1' then

            nextstate <= "00110";

        end if;

    when "00110" => -- request y input

        inputreq <= '1';

        if inputrec = '1' then

            nextstate <= "00111";
```

```
    end if;

when "00111" => -- get y input

    in_y <= input;
    en_y <= '1';
    nextstate <= "01000";

when "01000" => -- close y input

    en_y <= '0';
    inputreq <= '0';
    nextstate <= "01001";

when "01001" => -- wait for input

    if inputrdy = '1' then

        nextstate <= "01010";

    end if;

when "01010" => -- request z input

    inputreq <= '1';

    if inputrec = '1' then

        nextstate <= "01011";

    end if;

when "01011" => -- get z input

    in_z <= input;
    en_z <= '1';
    nextstate <= "01100";

when "01100" => -- close z input

    en_z <= '0';
    inputreq <= '0';
```

```
        nextstate <= "01101";

    when "01101" => -- wait state

        null;
        nextstate <= "01110";

    when "01110" => -- begin cordic interation

        in_x <= next_x;
        en_x <= '1';
        in_y <= next_y;
        en_y <= '1';
        in_z <= next_z;
        en_z <= '1';
        nextstate <= "01111";

    when "01111" => -- increment iteration

        en_x <= '0';
        en_y <= '0';
        en_z <= '0';
        iterate <= '1';
        nextstate <= "10000";

    when "10000" =>

        iterate <= '0';
        nextstate <= "10001";

    when "10001" => -- check for end of iteration

        if iteration = "0000" then

            nextstate <= "10010";

        else

            nextstate <= "01110";

        end if;
```

```
when "10010" => -- result buffer ready check

    if resultrdy = '1' then

        nextstate <= "10011";

    end if;

when "10011" => -- send x result

    result <= x;
    resultreq <= '1';

    if resultrec = '1' then

        nextstate <= "10100";

    end if;

when "10100" => -- close result request

    resultreq <= '0';
    nextstate <= "10101";


when "10101" => -- result buffer ready check

    if resultrdy = '1' then

        nextstate <= "10111";

    end if;

when "10111" => -- send y result

    result <= y;
    resultreq <= '1';

    if resultrec = '1' then

        nextstate <= "11000";
```

```
            end if;

        when "11000" => -- close result request

            resultreq <= '0';
            nextstate <= "11001";

        when "11001" => -- result buffer ready check

            if resultrdy = '1' then

                nextstate <= "11010";

            end if;

        when "11010" => -- send z result

            result <= z;
            resultreq <= '1';

            if resultrec = '1' then

                nextstate <= "11011";

            end if;

        when "11011" => -- close result request

            resultreq <= '0';
            nextstate <= "00000";

        when OTHERS =>

            nextstate <= "00000";

        end case;

    end if;

end process machine;

end;
```