A Framework for FPGA Functional Units in High Performance Computing

Andreas Koltes
Department of Informatics and Mathematics
University of Passau
Passau, Germany
Email: koltes@ieee.org

John T. O'Donnell
Department of Computing Science
University of Glasgow
Glasgow, United Kingdom
Email: jtod@dcs.gla.ac.uk

***Abstract*—FPGAs make it practical to speed up a program by defining hardware functional units that perform calculations faster than can be achieved in software. Specialised digital circuits avoid the overhead of executing sequences of instructions, and they make available the massive parallelism of the components. The FPGA operates as a coprocessor controlled by a conventional computer. An application that combines software with hardware in this way needs an interface between a communications port to the processor and the signals connected to the functional units. We present a framework that supports the design of such systems. The framework consists of a generic controller circuit defined in VHDL that can be configured by the user according to the needs of the functional units and the I/O channel. The controller contains a register file and a pipelined programmable register transfer machine, and it supports the design of both stateless and stateful functional units. Two examples are described: the implementation of a set of basic stateless arithmetic functional units, and the implementation of a stateful algorithm that exploits circuit parallelism.**

***Keywords*-FPGA; interface;**

I. Introduction

Many programs perform large numbers of timeconsuming operations. One way to run such programs faster is to split them into several tasks to be executed in parallel on different processor cores. Another approach is to make the basic operations themselves faster using hardware accelerators. One example of this is to provide floating point operations in hardware, rather then performing them in software.

Many computations can be performed faster by a specialised digital circuit than by a general purpose circuit (i.e. processor) running a program.

There are two fundamental reasons that circuits may be faster. The first is that the actual computation that is needed can be performed directly, without also requiring the overheads of fetching instructions, decoding them, and so on. For highly repetitive calculations, this can make hardware significantly faster than a corresponding program, and the hardware is relatively easy to design.

An even more fundamental factor is that digital circuits contain an extraordinary degree of parallelism. All the components operate in parallel, although the useful parallelism in a synchronous circuit is limited by the critical path depth. The ratio between the number of components and the critical path depth may be between 10^3 to 10^5. With careful circuit design, much of this large factor can be converted into useful parallelism.

Particular programs may require specialised operations, and it is impossible to support all of these in fixed hardware coprocessors. FPGAs offer the programmer the ability to define new hardware implementations of key operations used in a program. This makes it possible to use efficient hardware to avoid the overhead of executing sequences of instructions, and it offers an extremely high degree of parallelism.

Reconfigurable circuits, such as FPGAs, allow specialised circuit designs to be implemented quickly and cheaply [1] [2]. They offer the possibility of supporting slow operations in hardware at speeds much higher than can be achieved using standard processors. Reconfigurable hardware gives much of the benefit of fabricating a new circuit design at a much lower cost.

In order to use an FPGA to speed up a program, it is necessary first to identify a set of operations to be performed in hardware. These must be implemented as digital circuits, called *functional units*. Finally, an interface needs to be constructed that allows the processor to communicate with the new circuit.

Designing the interface is a significant challenge. It has to communicate with a processor, using an input/output channel, and it also has to communicate with a set of circuits via digital signals. The interface must handle the handshaking protocols required by the processor, as well as the buffering and timing requirements of the circuit. In some cases, it is useful for the interface to coordinate the operation of the functional units, treating them as microoperations in order to perform a larger calculation. To meet all these requirements, it is useful to organise the interface as a programmable register transfer machine (essentially a small RISC processor) with a register file. Furthermore, the interface cannot be a fixed circuit: parts of it need to be changed as the application circuits change.

This paper presents the design of a generic interface that

addresses these challenges. The interface is a digital circuit, defined in VHDL, that can be embedded on an FPGA along with functional units designed by a programmer to accelerate key operations. The interface circuit is a programmable register transfer machine, which can collect data from the processor, buffer it, run the functional units, obtain their results, and deliver them back to the processor. The work aims to improve portability, by providing a generic controller that can be adapted to a wide variety of computer systems.

The paper also discusses two distinct methods for using an FPGA: implementing stateless functional units, and implementing data parallel operations where the functional units hold persistent data in a state. We show how both methods are supported by the controller, and give an example of each.

The architecture of the controller is specified as a set of generics in VHDL. It contains several subsystems; some can be used without modification, while others are templates that will generate the actual circuits, under the control of parameters supplied by the user.

The controller and the case studies have been implemented and tested on an Altera Cyclone FPGA [3], using VHDL to specify the circuits. A complete description of the system, including full documentation of the interface and protocols, as well as the case studies, appears in Koltes' dissertation [4]. The dissertation also contains the VHDL code, and provides the information needed to use the system for practical applications.

Our results do not make the use of hardware accelerators as easy as ordinary programming. The user still needs to be able to design circuits as well as to write software. However, the work presented here does make the task significantly easier and more portable.

Several previous systems have used FPGAs to provide new operations to enhance a system's instruction set. Eisenring and Platzner present a theoretical model for describing such systems [5]. The CHIMAERA system [6] uses a processor tightly coupled to a reconfigurable array that implements operations used by the instruction set. The main difference with our work is that CHIMAERA is not a generic framework aimed at portability.

Wirthlin and Hutchins show how to use partial reconfiguration of an FPGA to allow an instruction set to be modified dynamically [7]. This is useful when the functional unit circuits require too much space to fit simultaneously in an FPGA, although the time required to load a new instruction (i.e. to read an functional unit circuit into the FPGA) is substantial. Related approaches are described in [8], [9] and [10].

One of the strengths of the framework presented here is its flexibility: it can work with a broad spectrum of microcontrollers and interconnection systems, and this does not require any modification to the processor architecture itself, while allowing custom instructions to be introduced directly into the microcontroller.

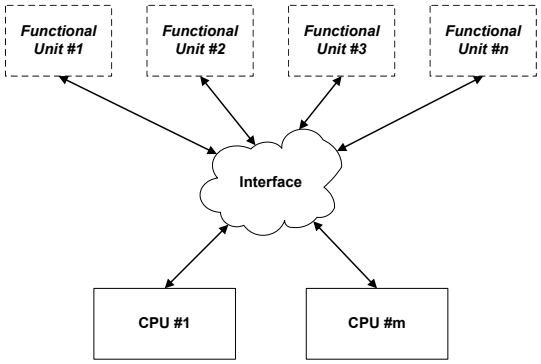

Figure 1. High level organisation. The main program is written in C or any other programming language, and runs in one or more CPUs which communicate via the interface with a set of functional units. The interface and the functional units are programmed using VHDL. The Interface comprises VHDL modules described in this paper, and the Functional Units communicate with the Interface according to protocols. The Interface can be configured by editing its VHDL definition.

Section II gives an overview of the system, discussing how the FPGA, the interface, and the CPU fit together. Section III describes the central component of the architecture, a Register Transfer Machine. This is a RISC processor that provides a register file and a simple instruction set. Section IV then discusses how the programmer can develop an application, for both stateless and stateful functional units. Section V concludes.

II. Overview of the framework

The programmer identifies a set of operations suitable for hardware implementation. These should have the following characteristics: they require a relatively long sequence of ordinary instructions to perform; they can be performed much more quickly using circuit techniques (e.g. by exploiting the parallelism inherent in circuits); they are executed frequently. The programmer then designs a dedicated circuit, called a functional unit, that implements each of the operations. Each functional unit is designed to interact with the central interface using a standard signal protocol, which is defined by the framework.

The aim is to speed up a program running on one or more processors by augmenting the processors with a set of *functional units*. A functional unit is a circuit that performs some computation significantly faster than can be done in software. The entire solution consists of a software component running in the processors and a hardware component comprising the functional units. Figure 1 shows the high level organisation of the system; the CPUs are in a standard computer while the functional units and the interface are embedded in an FPGA. The interface is generic, making it reusable across projects.

The interface needs to be able to execute instructions that

control the functional units. It also needs to retain information, enabling a sequence of functional unit operations to be performed, and to package operands and results according to the communications protocols.

These requirements are satisfied by organising the interface as a *register transfer machine*. This is a simple programmable datapath that contains a register file, and that has an instruction set for communications.

The entire system is controlled by the host computer. To perform an accelerated operation, the host sends one or more packets of data to the controller on the FPGA. The controller then coordinates the execution of the operations and returns the final results to the processor. From the processor's point of view, the FPGA acts like a coprocessor comprising one or more functional units. The host computer can send instructions to be performed on any of the functional units. Within the FPGA, the instructions may be executed out of order, but the stream of results returned to the processor will be consistent with the stream of instructions that were issued. This is similar to the effect of out-of-order execution within a sequential superscalar processor.

The register transfer machine communicates with the host processor using a transceiver circuit. There are many different physical interfaces that the FPGA might need to interact with. In some cases a predefined transceiver interface module may be available, and this can be combined with the VHDL definition of the controller. Depending on the system, it may be necessary to create a new transceiver circuit.

The controller is a digital circuit which is specified in VHDL, an industry standard language for designing digital circuits. The interface is customisable, and contains parameters that can be modified easily. For example, the word size used for the register file is adjustable, so the interface can meet the requirements of the functional units while requiring as small a portion of the FPGA as possible.

The system contains several units: the interface to the CPUs, the central control of the FPGA (a Register Transfer Machine), and the functional units (Figure 2).

Figure 3 shows the structure of the interface from the programmer's point of view. To use the system, the programmer needs to

- Partition the algorithm into a software part, to run in the processor(s);
- Define the specialised operations and implement them as functional units, using VHDL;
- Configure the interface framework by specifying size parameters for the register file, and selecting the appropriate transmitter and receiver modules.

III. Register transfer machine

The core of the interface is a register transfer machine (RTM). This is a microcontroller with a RISC style architecture, based on register files and instructions that act on

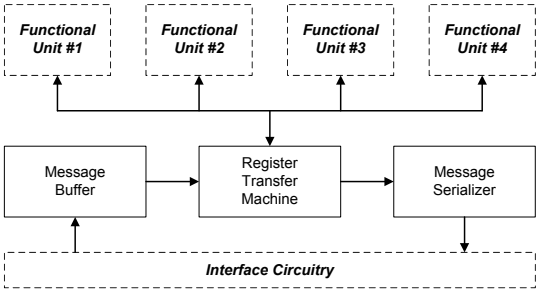

Figure 2. Structure of the system. The subsystems shown in this figure are specified in VHDL and run on the FPGA chip. The interface circuitry is a low level transceiver that communicates directly with the port pins on the chip. Incoming and outgoing messages go via hardware buffers to the central Register Transfer Machine, which controls the functional units.

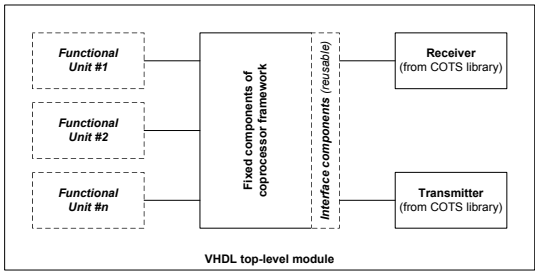

Figure 3. Top level VHDL module. The main components of the system are shown from the programmer's point of view. The receiver and transmitter come from libraries. The main Register Transfer Machine is a fixed VHDL definition, although it has a number of size parameters. The functional units are specific to the application.

the registers. The architecture contains two register files. The main register file holds data, and its word size is configurable in multiples of 32 bits. There is a secondary register file holding vectors of flags, which are often useful for controlling the functional units. The RTM instructions may have up to three operands to be fetched from the register file, and up to two results may be loaded into the register file.

The RTM interacts with the host computer through a message buffer for input and a message serialiser for output, and it interacts directly with the functional units using digital signals). The message buffer and serialiser communicate with the host using standard FPGA circuits from a COTS library.

The register transfer machine executes instructions using a pipeline (Figure 4), in order to attain concurrency among the instructions and to reduce the clock period. The pipeline was designed with most registers at the end of the pipeline stages, because most FPGAs have their registers after the function generators. Handshaking is used to control transmission of data between pipeline stages. This allows local control to

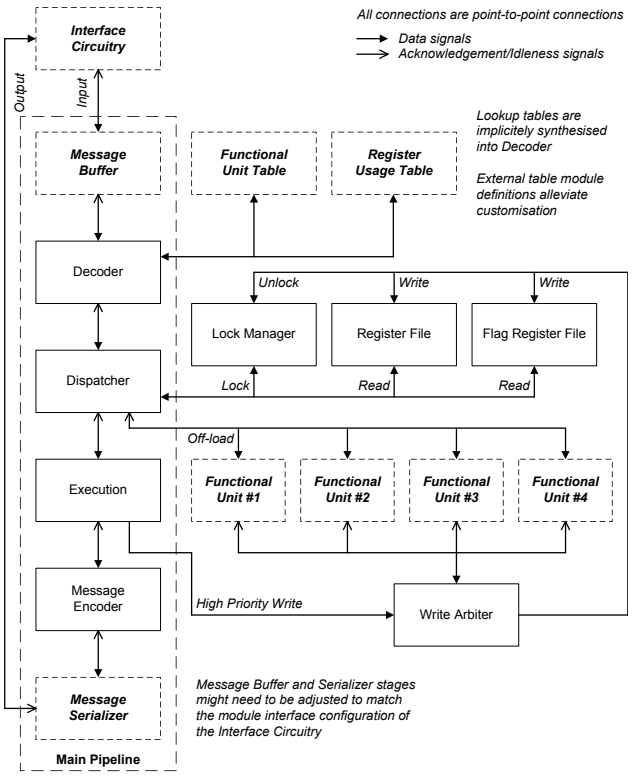

Figure 4. Organisation of Register Transfer Machine. The RTM is a RISC processor with a pipeline (the column on the left side of the figure) comprising a decoder, dispatcher, and execution stage. The processor keeps state in a register file, which provides a source and sink for data transmitted to the functional units.

stall the transmission when necessary; there is no global control for stalling the pipeline. The pipeline contains the following stages:

- *Message buffer.* The first stage receives data from the FPGA input port connected to the host processor, and converts it to a form usable by the decoder. This stage needs to be implemented according to the communication protocol used by the host processor.
- *Decoder.* The current instruction is decoded into a vector of signals that control the execution stage.
- *Dispatcher.* Reads from the register file take place in the dispatcher stage, and instructions that initiate a functional unit operation transmit data to the functional unit through a register in this stage.
- *Execution.* Instructions that operate on the state of the RTM are executed.
- *Message encoder.* There are several types of message that can be sent from the RTM to the host, including data records and flag vectors, and these are multiplexed into a single standard vector of signals.
- *Message seraliser.* The signal vector is converted to the form required by the communication port to the host, and is transmitted on the port.

The speed of the system is determined by two factors: the latency of the communication interface to the host computer, and the clock speed of the FPGA. Our implementation used a prototyping board which is intended for experimentation and software development, but not for high speed. In particular, only a very slow connection from the FPGA board to the processor was available. However, this is not a limitation of the approach: there are FPGAs that are tightly integrated with processors, offering extremely high transfer rates. In such a system, the main limitation on performance would be the speed of the circuit on the FPGA.

The generic controller is designed to minimise the clock period; this is achieved by pipelining, so the critical path in the controller is short. In general, FPGAs have slower clocks than processors, and the RTM controller should allow the fastest clock speed that the FPGA allows. The main limitation on performance will be the functional unit circuits.

IV. Developing an application

The main task for the programmer is to design the functional units. They must interact with the controller according to the framework's protocol, but apart from that requirement, the designer has complete freedom in the internal structure of a functional unit.

Figure 5 shows the architecture of a minimal stateless functional unit. The purpose of the unit is to perform a calculation, which is implemented by a black box circuit. The unit interacts with the controller according to the protocol, which is documented in detail in [4].

An application program running on a host computer uses the FPGA, with its functional units, similarly to the way it would use any conventional coprocessor, such as a dedicated floating point unit. Naturally there will not be an instruction in the processor's instruction set that uses the newly created operation. Typically the FPGA would be treated as a fast I/O device. The mechanism for executing an operation in a functional unit depends on the system, but in general it would be the same as for any other coprocessor operation.

The interface framework allows several functional units to be incorporated on the FPGA, and these units may have different designs. Thus it is possible to provide a set of operations.

Each functional unit interacts with the register transfer machine according to a protocol expressed as a finite state machine (an example is shown in Figure 6). The register transfer machine has an instruction set that is used by the programmer to control transmission of data between the registers to the functional units.

There are two major classes of functional units: *stateless* and *stateful*, discussed in more detail in the following sections.

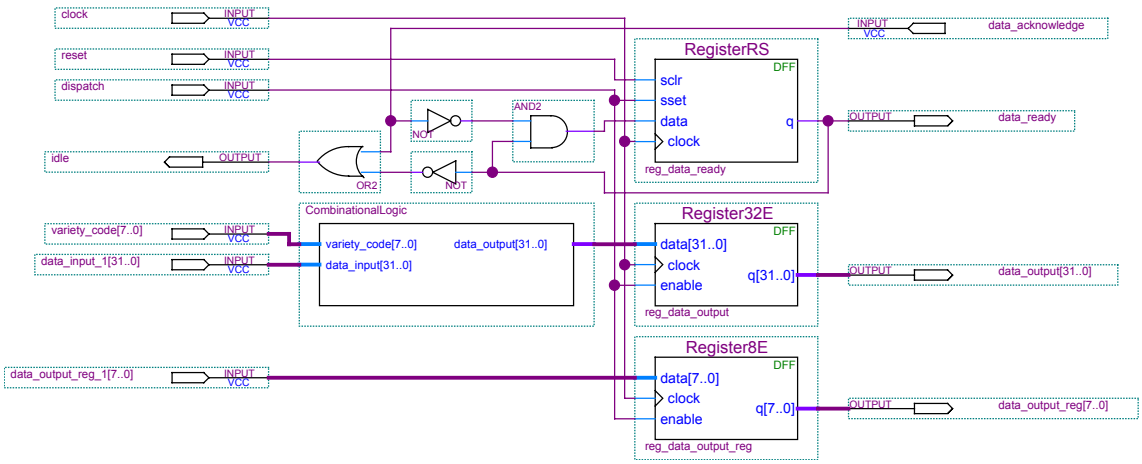

Figure 5. Minimal functional unit. This is an example of a functional unit circuit, showing the signals that connect it to the controller. The circuit computes a pure Boolean logic function, using logic gates. A real functional unit would have a similar interface to the controller, although there would normally be more signals, and the internal computational circuitry would be much more complex.

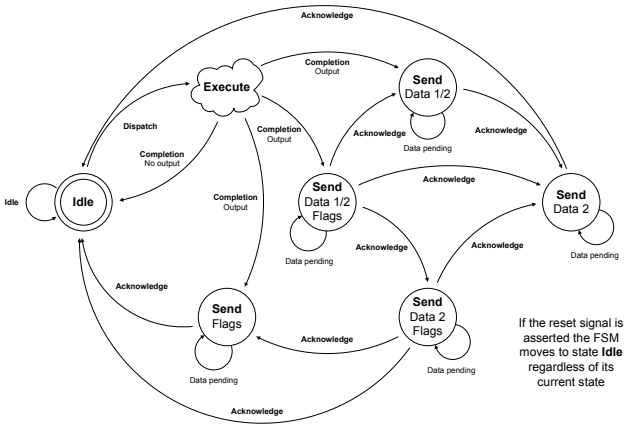

Figure 6. Example of a finite state machine for functional unit. Each functional unit communicates with the RTM controller according to a fixed protocol, which is implemented within the functional unit by a finite state machine. The FSM coordinates the transmission of data, and may also control the datapath within the functional unit.

A. *Stateless functional units*

A stateless unit computes a pure function of its operands. Once it transmits its result to the controller, the unit contains no memory that will affect future computations. Examples of stateless functional units are arithmetic units, trigonometric function calculators, etc.

As a simple example, consider a set of functional units to perform a family of arithmetic operations on integers. (The full details appear in [4].) This example is chosen for simplicity, and to test and measure the system; in a real application it would be worthwhile designing functional units only for operations that are significantly more time consuming.

The programmer needs to decide on the set of operations, design the functional units, and specify a set of instructions for the RTM controller to perform the operations. For this example, the hardware design is straightforward; the circuits are standard, and VHDL can synthesise them from standard notation, much as a compiler can generate machine language from similar notation. For more complex operations, it may be challenging to design the functional units, just as programming may be challenging for hard problems.

Figure 7 shows the instruction set architecture for a stateless functional unit. The instructions follow the formats allowed by the RTM controller, and are similar to arithmetic instructions on a typical RISC processor. Each instruction specifies the operation, the operand registers, and the result registers.

B. *Stateful functional units*

A stateful unit has a local persistent memory. Operations performed by the unit may depend on data in the memory, may modify it, and may return part of it to the controller. Examples of stateful functional units are histogram calculators, pseudorandom number generators, and associative memories.

We have developed an application that uses a stateful functional unit to implement an algorithm that performs simple computations in parallel on every element of a data structure. With conventional data structures, the processor performs operations on one element at a time, leaving the remainder of the data structure inert. The approach used here is to use circuit parallelism to provide a richer set of primitive operations.

The spplication is an implementation of the χ-sort suite

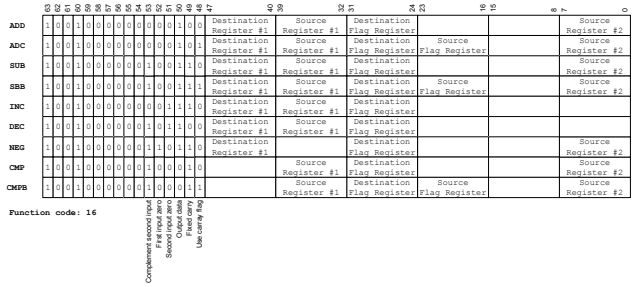

Figure 7. Instruction set architecture for stateless functional units. The instructions shown here follow the instruction format for the RTM controller. To execute the instructions, the controller obtains the operands from the register file, dispatches the operations to suitable functional units, receives the results, and places the results into the register file. The operations (addition, subtraction, etc.) correspond to the functions computed by the functional units.

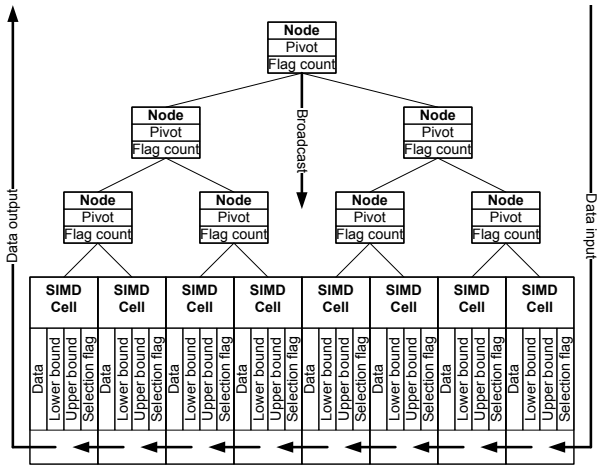

Figure 8. Organisation of stateful functional unit for the XI algorithm. The functional unit is organised as a binary tree of interior node circuits and leaf cell circuits. The persistent state is distributed across the cells, while computations are performed in both the cells and the nodes. The leaf "cell" processors provide permanent storage and perform comparisons on indices; the interior nodes do not have persistent state, but they do contain simple combinational logic functions that implement parallel scans and folds required by the algorithm.

[11], which performs selection and sorting using using an array represented with *index intervals*. With ordinary arrays, an element is identified by a index. With the index-interval representation, an approximate index can be specified. An element with index interval $\langle p, q\rangle$ belongs in the array at some index i such that $p \leq i \leq q$. An initial array represents the complete lack of knowledge of where the elements belong by assigning each element an index interval $\langle 0, n-1\rangle$.

In sequential algorithms the data structures can be modified only one element at a time as the processor executes load and store instructions. With circuit parallelism, data structures can be active. Each element of the array is stored in a small processor called a *cell*, which is implemented as small circuit in the FPGA. Cells contain combinational logic as well as storage; thus cells are a form of "smart memory". This capability enables the χ-sort algorithm to recalculate the index interval of every data item in parallel, at clock speeds.

The χ-sort algorithm executes in the Register Transfer Machine, which issues microinstructions to a stateful functional unit, whose organisation is shown in Figure 8. The functional unit is a tree network with leaf cells containing persistent memory, and interior node circuits that provide communications and support parallel folds and scans on associative operators.

The cell circuit contains a small amount of storage, enough to hold one data element and its index interval. The cell also contains a simple arithmetic circuit that can perform comparisons and additions. The entire set of cells form a smart memory that implements a microinstruction set specifically targeted at the χ-sort algorithm. The RTM implements operations (e.g. performing a selection operation) by issuing a set of microinstructions to the cells. Figure 9 shows the implementation of the cell circuit.

Circuit parallelism enables χ-sort to execute significantly faster than can be achieved with software on a conventional processor. Each operation takes a fixed number of clock cycles with the FPGA; with a CPU each operation requires an iteration that takes time proportional to the number of data elements.

The algorithm has been implemented on an Altera FPGA, a small scale system intended for prototyping and software development with a clock speed of approximately 50Mhz.

V. Conclusion

Several factors make it challenging to use FPGAs in ordinary programming. The solution requires circuit design skills as well as programming skills, an overall structure has to be found for the FPGA circuit, an infrastructure is required for holding data on the FPGA and delivering it to the functional units.

We have presented a framework that addresses the interfacing issues in using FPGAs. It provides an efficient register transfer machine for coordinating the data transfers and controlling the functional units, relieving the programmer from reinventing a significant amount of circuitry. The framework is implemented in VHDL, with full documentation. To use it, the programmer needs to configure the interface (by making some VHDL definitions) and to define the functional units.

The most complex details of the interfacing are provided by the framework; the programmer's task is to design the core logic of the functional unit (hardware design, using VHDL) and to program the controller (which is software design, and considerably simpler than it would be to design a dedicated interface from the ground up).

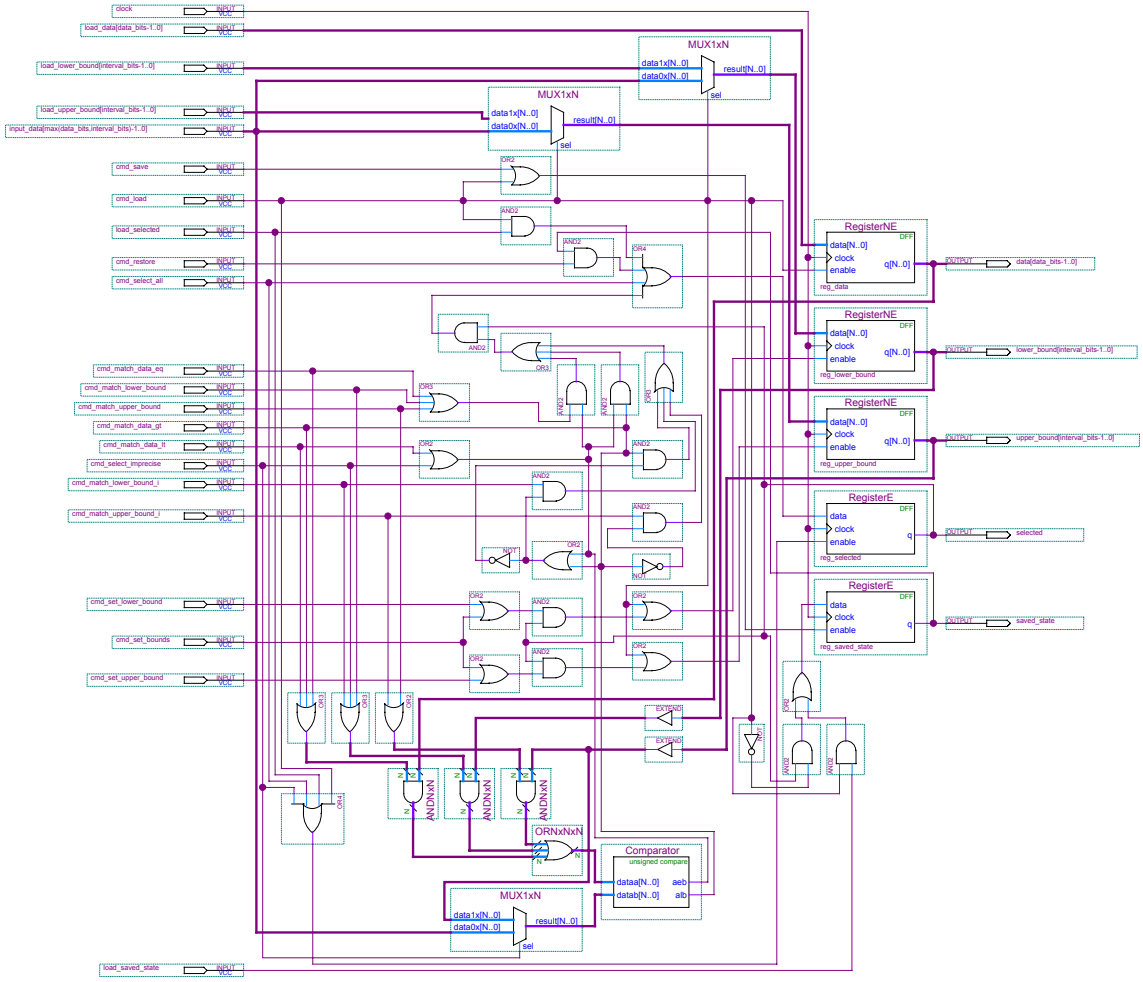

Figure 9. Cell circuit for XI algorithm. A cell corresponds to a word of memory, but it contains a small amount of computatational hardware as well as storage. There is an array of cells, providing a memory that can hold an array. The entire set of cells comprises an extremely fine grain data parallel architecture, which is targeted specifically to the χ-sort algorithm. The programmer begins by defining the behaviour of the high level operations in the algorithm; these perform the same operation simultaneously in every cell. Next, a circuit is designed that provides both the storage and computation required for every data element. Finally, this circuit is specified using VHDL. The figure shows the low level layout defined in the VHDL design.

The largest remaining challenge is the expertise required to define new functional units. Much progress has been made in high level hardware description languages and hardware synthesis, but for the foreseeable future it will be harder and require more knowledge to put part of an algorithm into an FPGA rather than treating it as pure software. However, the efficiency and parallelism offered by digital circuits are very large, so this effort is likely to be justified for many demanding applications.

REFERENCES

[1] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, 2002.

[2] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung, "Reconfigurable computing: architectures and design methods," *IEEE Proc. on Computer and Digital Technology*, vol. 152, no. 2, pp. 193–207, March 2005.

[3] *Cyclone Device Handbook, Vol. 1*, Altera Corporation, 2008.

[4] A. Koltes, "A flexible architecture framework for FPGA based coprocessors," University of Passau, Faculty of Computing Science and Mathematics, Innstr. 44, D-94032 Passau, Germany, Tech. Rep., 2008, Diplom Thesis.

[5] M. Eisenring and M. Platzner, "An implementation framework for runtime reconfigurable systems," in *Proc. 2nd Int. Workshop on Engineering of Reconfigurable Hardware/Software Objects (ENREGL00)*, June 2000, pp. 151–157.

[6] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banergee, "CHIMAERA: a high performance architecture with a tightly coupled reconfigurable functional unit," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 225–235, 2000.

[7] M. J. Wirthlin and B. L. Hutchings, "A dynamic instruction set computer," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'95)*, 1995, p. 0099.

[8] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction set metamorphosis," *IEEE Computer*, vol. 26, no. 3, pp. 11–18, March 1993.

[9] A. DeHon, "DPGA-coupled microprocessors: commodity ICs for the early 21st century," in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, April 1994, pp. 31–39.

[10] R. Razdan and M. D. Smith, "A high performance microarchitecture with hardware programmable functional units," in *Proc. 27th Annual Symposium on Microarchitecture*. ACM, 1994, pp. 172–180.

[11] J. O'Donnell, "Functional microprogramming for a data parallel architecture," in *Proc. 1988 Glasgow Workshop on Functional Programming*. Department of Computing Science, University of Glasgow, 1988, pp. 124–145.