TDT4720 Datamaskinkonstruksjon og -arkitektur, fordypning

Høsten 2005

Introduksjon til bruk av FPGA i vitenskaplige beregninger

Lars Krutådal

Faglærer: Jørn Amundsen

LIST OF FIGURES

LIST OF TABLES

Abstract

This text is aimed at giving an introduction to the use of Field-Programmable Gate Arrays (FPGAs) in scientific high-performance computing, with particular focus on the use of the Cray XD1 supercomputer. FPGAs in general are introduced, followed by information to guide the creation of software for the Cray XD1 and attached FPGA modules, as well as information regarding the creation of the actual logic for the FPGAs.

Preface

This introduction to FPGA-accelerated high-performance computing on the Cray XD1 supercomputer was written as an assignment in the course TDT4720 at the ninth semester of a five year Master course in computer science at the Norwegian University of Science and Technology (NUST/NTNU). It was primarily guided by Jørn Aslak Amundsen, associate professor at the Division of Complex Computing Systems.

Several people have contributed in some way or another to the work involved in putting together in this text, and I would like to thank everyone involved, in particular the following people:

Jørn Aslak Amundsen who, as mentioned, was the guide for this project, and has contributed with opinions and guidance at fixed weekly meetings, as well as providing equipment and contacts whenever it has been required.

Arve Dispen, who is the administrator of most of the HPC systems at NUST/NTNU including the Cray XD1, for being quite patient with the large number of system failures induced by the activities involved in writing this text, and being very helpful with organizing various hardware and configuration support to allow the activities to take place.

Gunnar Tufte, who has acted as a secondary advisor in this project, and in particular has made several corrections and suggestions in the sections dealing with the technical aspects of the FPGA.

Geir Kjetil Sandve and Finn Drabløs, who organized and lead several meetings during the semester with focus on writing a proper report, and also made several suggestions to content inspired from their own line of work, bioinformatics.

Jan Christian Meyer and Gunnar Lien, who provided hardware needed for some of the debugging activities.

Tarjei Sveinsgjerd Hveem and any other people I forgot, who read and commented on early drafts of this text.

1 Introduction

In the summer of 2005, a Cray XD1 supercomputer was acquired as a resource for the bioinformatics group, as a part of the cluster infrastructure at the Norwegian University of Science and Technology (NUST/NTNU). The ultimate goal was to explore using Field-Programmable Gate Arrays (FPGAs) as a way to increase the execution speed of applications, by using the customizable nature of the FPGA to perform parallel parts of the computations on the FPGAs instead of on standard CPUs.

This project is based around the Cray XD1, and focuses on gaining a high degree of familiarity with the platform. The end result, in form of this text, is an introduction to the use of FPGAs from scientific programs in general, and the Cray XD1 in particular.

This text is divided into three major parts. It opens with an introduction to FPGAs in general, including traditional uses of the FPGA and alternatives to FPGAs in the field of application acceleration. The second part is geared towards application developers, with a focus on how to access and use the FPGA from C applications. This part also includes an introduction of how to combine OpenMP and MPI with the FPGAs. The third part considers the hardware side, with a focus on creating and debugging FPGA designs on this particular platform.

For the reader who is not familiar with FPGAs in general, and only need to code a program that uses an existing library on the FPGA, the recommended path is to start with Section 2, 3 and 4 for an introduction to the concept, as well as a discussion of different platforms and some concrete implementations. These sections can however be omitted, but at the loss of some understanding about how the FPGA works. After understanding the basics, Section 5 and 6 are then recommended for information about the coding process. The remaining sections largely concern hardware-specific development, and are not vital to the software-focused developer.

For the reader who is familiar with FPGAs, but not with using it in a HPC or application acceleration environment, Section 4, the discussion of various HPC-related FPGA implementations is recommended as the starting point. Section 5 then studies the API of the Cray XD1, and should be studied, but Section 6 covers higher-level distributed programming and can be mostly skipped with the possible exception of the parts regarding FPGA-side access arbitration. Section 7 and 8, which cover the development of FPGA logic, should be studied last.

For the reader who is both familiar with FPGAs and with using it in a HPC context, much of the material in this paper should be familiar, and only the parts particular for the Cray XD1 platform, namely Section 5 and 7, are likely to contain significant new material.

Finally, the reader who is not familiar with FPGAs, but still wants to create FPGA logic to run programs on the Cray XD1 platform, should be warned that this is not an easy task. All sections should in this case be studied in order, but the foundation knowledge and

skills for FPGA development is not included here, and must be obtained elsewhere. It is indicated in the text where this applies, and suggestions for additional reading material should be listed where appropriate.

I Introduction to FPGAs

This part will serve as an introduction to FPGAs in general. It opens with the basics of the FPGA in Section 2, which includes a short technical introduction, and discusses various applications for the FPGA. It continues to explore a few different platforms that employ FPGAs for this use in Section 3, and rounds off with a selection of contemporary software powered by such platforms in Section 4.

2 The basics of the FPGA

It can be assumed that some users of FPGA application accelerator processors, who focus mostly on the applications and are satisfied with pre-made libraries, will not be overly concerned with the inner workings of the FPGA. In Part II, this is largely assumed to be unknown. However, for users who have planned to start creating their own FPGA designs, this is useful knowledge. This text will not give you the foundation to jump directly from a blank state in hardware design to creating complex circuits on the FPGA, as this material can easily fill several books by itself, but it will hopefully work as an introduction to the basics, as well as a guide to the particulars of development on the Cray XD1 for new and experienced hardware developers.

This section will give a summary on the general workings of FPGAs, starting with a general technical introduction and the fields where FPGAs are typically employed. Drawbacks of FPGAs will also be discussed, followed by a discussion of using FPGAs as application acceleration processors. Finally, it will be compared to other technologies that can, to varying extents, be used for application acceleration in similar ways.

2.1 Technical Introduction

A *Field-Programmable Gate Array* (FPGA) is a reprogrammable semiconductor device which can be adapted for a variety of uses. As the name indicates, a FPGA can be programmed “in the field”, allowing a high degree of flexibility at certain costs, discussed later in Section 2.3. This stands as a contrast to its traditional counterpart, the *Application Specific Integrated Circuit* (ASIC), which cannot be altered in the same way.

The central part of the FPGA are the *Configurable Logic Blocks* (CLB) which are connected through a set of routing channels to special programmable switch matrices. The exact content and technology used in the CLB differs between vendors and even particular FPGA implementations[1], but they all have the purpose of providing

[1] Three common technologies for the FPGA itself are SRAM (not to be confused with data storage), Antifuse and E^2PROM/FLASH. These differ in many aspects, such as in how they are programmed (SRAM and some E^2PROM/FLASH devices can be programmed while resident in the system, while Antifuse needs to be fitted into a special device programmer), power consumption (significantly lower for

programmable combinatorial and synchronous logic. The switching matrices also connect the CLBs to a set of *Input/Output Blocks* (IOB) used by the FPGA to communicate and interact with external circuitry and devices. Most FPGAs also have specialized circuitry used to perform common operations, such as BRAMs and multiplicators, and in some cases even have one or more integrated CPUs. The CLBs, IOBs and other modules can be combined and connected in a variety of ways using the underlying programmable switch matrices, allowing the FPGA to implement any logic that is within its technical size and routing boundaries [MANO01 pp. 326-333]. The Xilinx FPGA that comes with the Cray XD1 has several specialized modules ranging from bit matrix multiplication to clock signal manipulation, as well as two integrated CPUs; see Appendix A.2.3.

A conceptual, scaled-down illustration of these concepts can be found in Figure 2.1, where a number of CLBs (centre) and IOBs (outer edge) are connected by a number of switch matrix elements. Note, however, that the switches become static links that are set up between the various elements when the FPGA is initialized with a design, and the switching in the illustration should therefore not be interpreted as dynamic. The actual number of wires that connect the different elements and the number of connections that can pass through a switch also vary significantly between implementations.

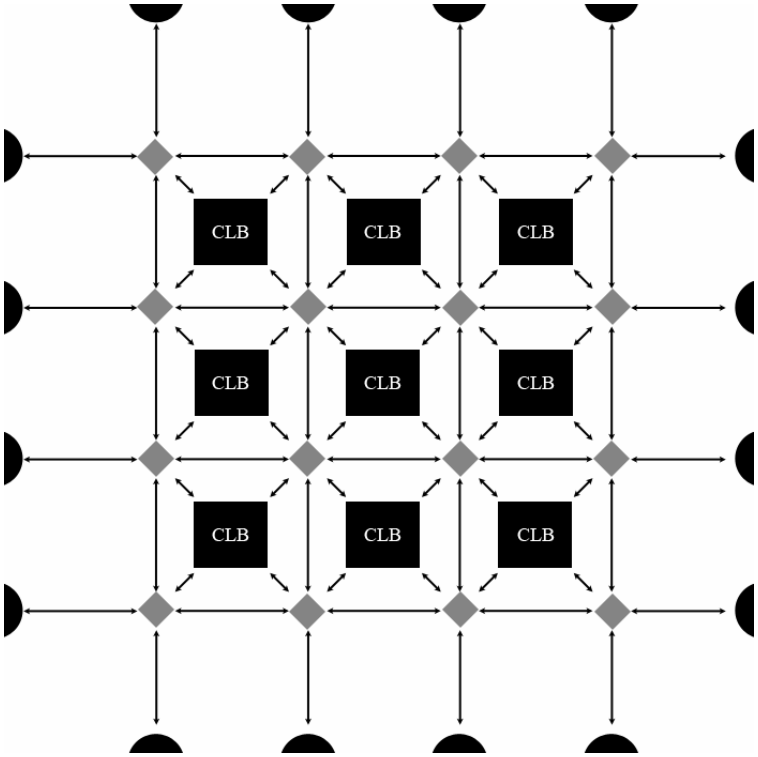

Figure 2.1: A conceptual figure of a FPGA interior.

Antifuse than the two alternatives), and volatility (SRAM needs to be reprogrammed on each power-on). SRAM is the most common of these three, while the other two are most commonly found in special applications (for instance, Antifuse is by nature more resilient to the effects of radiation, and is therefore common in aerospace applications). A similar aspect of differing CLB architectures are MUX-based (multiplexer) versus LUT-based (lookup table) logic blocks, where LUT-based designs are the most common today [MAX04 pp. 9-23, 57-77].

The complexity and size of the logics that can be implemented on a FPGA is dependent on the number and type of available CLBs, IOBs and other modules, as well as the sophistication of the so-called *synthesis*[2] and *place-and-route*[3] tools used to transform a chip design to a binary file that can be loaded to the FPGA. The utilization of the various special functional modules mentioned earlier depends on the cleverness of the synthesis tools, but they can also be assigned manually by the chip developer in most cases.

A detailed discussion of particular FPGA implementations is beyond the scope of this text, but official documentation is ample. Xilinx provides a wealth of information about its FPGAs at [XILIB], as does Altera at [ALTLIB]. [MANO01] discusses the structure and implementation of the Xilinx XC4000 FPGA, including detailed diagrams of the CLB and switch matrix internals. [MAX04] gives a more general description of the various implementation solutions, and how they've changed with time.

The FPGA technology is a subset of *Field Programmable Devices* (FPD) which also consists of the similar technologies *Simple-* and *Complex Programmable Logic Device* (SPLD/CPLD). The main difference between FPGAs and PLDs is that the latter only provides combinatorial logic, while FPGAs also offer sequential logic such as flip-flops. Information about PLDs can be found in [MANO01] and [PAT05 pp. B76-B77], as well as at [WIKI02].

2.2 Traditional uses of the FPGA

Due to its inherent flexibility, FPGAs have been deployed in a variety of fields ranging from the classical usage as a tool for prototyping, to being used as a co-processor to speed computing, to it itself being a system-on-a-chip (SoC). In many fields it is a direct competitor to the ASIC, where factors such as limitations on power consumption, production cost versus quantity and maintainability decide which of these alternatives fits the purpose. Certain parts of the spectrum can be filled by either one of the alternatives, like controllers for various products. Others can only easily be filled by one or the other, with FPGAs being the only viable choice for early chip prototyping and the best choice for devices produced in limited quantities, while ASICs still are the best suited for low-power compact chips in portable units as well as in most mass-produced devices. The rationale behind these claims is given in Section 2.3.

The classical raison d'etre of the FPGA is its use as a prototyping tool for ASICs. After the initial creation and simulation of a chip design, it can easily be uploaded to a FPGA, which is then connected to a host circuit for testing and verification. If any errors are detected, the design can be modified and reuploaded for further testing without much additional cost. If similar testing were to be performed by creating an ASIC for each step of the verification process, costs would be prohibitively large. The so-called non-recurring engineering cost (NRE) to setup a fab for a test run production of contemporary

[2] The *synthesis* of a FPGA design involves translating the VHDL code to a circuit, resulting in a *netlist*.
[3] The *place-and-route* process assigns the different circuit parts in the *netlist* to specific parts on the FPGA.

ASICs often runs into hundred of thousands of euro, making FPGAs a very attractive choice except for in the very latest stages of testing.

Another popular use for FPGAs is in devices that are produced in limited quantities. Again, the cost of the ASIC makes it prohibitively expensive to make a limited number of chips, while the FPGA with its higher per-unit cost but nonexistent NRE can be used for arbitrarily small quantities at reasonable cost. However, as soon as the number of devices reaches a certain number, often estimated at around 10.000, the per-unit cost of the FPGA is high enough to offset the high NRE of the ASIC.

Finally, there is the current issue of using FPGAs as platforms for reconfigurable computing, where the FPGA is either the central chip in a system and is reconfigured externally for particular uses, or serves as a co-chip in a larger system where it can be reconfigured on the fly by its host. It is particularly suited for algorithms that can make use of the parallelism offered by the FPGA architecture, and has wide applications in fields such as digital signal processing, cryptography and bioinformatics. This particular usage is further studied in Section 2.4.

A case could be made toward using FPGAs in consumer devices to allow future hardware upgrades, but this is usually considered overkill, as a cheaper ASIC with upgradeable firmware often has a level of configuration that is high enough for this use. While it will not reach the configurability of a FPGA, such devices rarely undergo fundamental changes after deployment, making the extra configurability unnecessary.

2.3 Weaknesses of the FPGA

The largest merit of the FPGA is, as mentioned earlier, its flexibility. It can be used in many fields where one would usually employ an ASIC, as well as in some where ASICs are not suitable, such as prototyping. This flexibility and reconfigurability does however come at a cost. In general, FPGAs have three traditional disadvantages compared to ASICs: die size versus logic complexity, speed, and power consumption.

Fitting a complex design on a FPGA will always take up more room than fitting the same design on a comparable ASIC. The reason for this should be obvious; while an ASIC can implement a design using a minimal number of gates and wires, a FPGA has to fit the design to a pre-made arrangement of logic blocks and switches. Another problem is that if certain kinds of combinatorial logic are predominant in the design, the FPGA may have to use a large number of logic blocks even if each block isn't fully utilized. This leads to a larger die size which increases the production cost of FPGAs, both through a lower yield and a lower number of chips per wafer. These increased production costs makes FPGAs more expensive than ASICs for large production runs, which possibly is the main reason ASICs are predominant in mass-produced electronic devices.

Another issue is speed, or more precisely, clock frequency. Most FPGAs today operate at speeds that are an order of magnitude lower than the higher-end ASICs, and while this

doesn't automatically translate to an order of magnitude lower performance, it does have an impact on the use of FPGAs in circuitry where high performance is critical. The main reason for this difference in speed lies in the design. FPGAs use a relatively higher minimal amount of transistors to perform the same logic function, both in the logic blocks and in additional transistors needed for the switching matrix, which causes a longer *critical path*[4]. To make up for the longer transition delay, the clock frequency has to be reduced.

While it is hard to predict the future, trends indicate that the difference in speed between FPGAs and ASICs could close somewhat. Clock frequency on commodity CPUs using current technology have reached a ceiling imposed by heat constraints, with certain models dissipating as much as 150W of power. FPGAs have not yet hit a similar ceiling, and therefore have a potential of gaining some ground on their ASIC counterparts, but due to the technology used, it is highly unlikely to overtake ASICs completely.

Finally, there is the issue of power consumption. FPGAs typically consume somewhat more power than equivalent ASIC counterparts, particularly true for SRAM implementations. The obvious reason for this is that, provided they use the same transistor technology, the total *leakage current*[5] will be greater in the FPGA, due to the larger number of transistors employed to solve the same task. Another source of additional power consumption lies in non-utilized or idle parts of the FPGA. While an ASIC includes only the transistors and circuitry actually used for the design, the FPGA often have large areas on the chip that do not perform any actual work. The amount of power actually consumed here varies between FPGA implementations.

2.4 Using FPGAs as Application Acceleration Processors

One of the more recent areas where FPGAs have started gaining popularity is in application acceleration. The idea is to use one or more FPGAs as co-processors in a larger system, where the host system can reconfigure the FPGAs at will to make them perform tasks or parts of tasks given to the host.

The speed issues mentioned in Section 2.3 place some restrictions on the nature of the tasks that can be performed efficiently on the FPGA. Purely sequential tasks are usually better performed on traditional CPUs. The same can be said for tasks that require a large amount of communication compared to the amount of computation involved, since the FPGAs typically have somewhat slower memory bandwidth than the CPU, due to their inherent speed limitations.

---

[4] The *critical path* is the path through the circuitry with longest delay. This is usually the path with the highest number of transistors, but the length of the wires connecting them also make an impact.

[5] *Leakage current* is a term which in this context is used for the power consumed and dissipated as heat by the transistors in a circuit. Traditionally, the power consumed by idle or non-switching operation (*static power*) has been relatively negligible, with a much larger amount of power being used when the transistor switches between its low and high states (*dynamic power*), but with contemporary 90nm chip processes, the static component has become more dominant [ACTELPWR, WIKI03].

The field in which FPGAs can be applied is first and foremost highly parallel algorithms, where operations can be independently performed on a large number of data elements on separate computational cores. The sequential nature of the CPU forces it to do these operations on one element at a time, while the FPGA can employ a large number of cores to parallelize the operations on these data elements, and then combine and process the results in arbitrary ways before transmitting them back to the CPU. The CPU is usually free to perform other tasks while it waits for the results to be returned from the FPGA. Applications of this kind range from search, for example through DNA sequences, to high-grade encryption and decryption.

Even if the algorithm is not highly parallel, the FPGA can be successfully applied in certain situations by running routines that are only loosely connected to the main parts of a program, in particular if these are computationally intensive and do not require much communication. One example of this could be an advanced random number generator, where the FPGA could buffer up a number of random numbers directly to the CPU memory which are then ready whenever the CPU requests them, freeing the CPU from the load of calculating them itself.

2.5 Alternatives to FPGAs for Application Acceleration

While FPGAs are a flexible choice for application acceleration, there are several other technologies, both traditional and new, that can be used to speed the execution of programs. These vary in that some of them are simply specialized instructions added to a standard sequential processor, while others are based on completely different ways of executing programs. This section will first discuss vector processors, and goes on to discuss the idea of adding vector arithmetics to standard sequential processors. Finally, the *Cell Microprocessor*, which incorporates many of these ideas, will be discussed

2.5.1 Vector Processors

One of the earliest forms of parallel computing was *Single Instruction Multiple Data* (SIMD) architectures in forms of *vector processors*, where work is done on a number of elements in parallel, instead of on one element at a time as is the case in sequential processors. Work on vector processors began in the 1960s, and the first working vector computer, the ILLIAC IV, was delivered in 1972, but performance was far behind what was expected. The most famous of the early vector-based computers, and possibly the earliest example of a commercially and technically successful vector computer is the *Cray-1* [WIKI04]. One example of a contemporary vector computer is the NEC Earth Simulator [KRUT04], which at its launch was the world's fastest supercomputer, but as of June 2005 holds the fourth place on the Top 500 list[6].

[6] http://www.top500.org

2.5.2 SIMD Extensions

While dedicated vector processors are not commonly used today, typically sequential processors have taken a leaf from the book of vector processors and introduced dedicated *SIMD Extensions*. As the name indicates, these are instructions that work on sets of data elements in parallel. Two of the first implementations, both of which shipped in 1995, are the *Visual Instruction Set* (VIS) on the Sun UltraSPARC [SUNVIS], and the *Multimedia Acceleration eXtensions* (MAX) on the Hewlett-Packard PA-RISC.

One of the most famous set of SIMD extensions was introduced with the Intel Pentium's 1997 Tillamook (P55C) revision as *MMX*, which is frequently rendered as *Matrix Math eXtensions* but officially is meaningless initialism. MMX reuses the 64-bit floating-point registers of the processor by partitioning them into 2x32, 4x16 or 8x8 bits, and uses arithmetic operations between two different registers, where an operator is applied separately on these bit sequences. MMX had several problems, the most significant one being that MMX and floating-point operations could not be performed at the same time or interleaved as they shared registers, incurring a penalty for switching modes [INTELMMX].

MMX has later been improved and greatly extended with the introduction of *Streaming SIMD Extensions*; SSE (with the Pentium III), SSE2 (with the Willamette Pentium 4) and SSE3 (with the Prescott Pentium 4). These provide a variety of SIMD instructions, but introduced eight new 128-bit registers, and therefore do not have the inherent weakness of sharing registers with the floating-point unit[7] [INTELSSE]. AMD also entered with its 3DNow! for the K6-2 in 1998, and an updated version was released with the Athlon XP. 3DNow! was a direct extension to MMX that allowed for SIMD floating-point operations[8] [AMD3DN].

2.5.3 The Cell Microprocessor

Vector processors could possibly make a comeback with the introduction of the *Cell Microprocessor*, a joint development project by IBM, Sony and Toshiba. The first-generation Cell is based around eight *Synergistic Processor Elements* (SPE), controlled by a Power Architecture compliant *Power Processor Element* (PPE). A high-bandwidth *Element Interconnect Bus* (EIB) connects the PPE, the SPEs, a memory controller and an I/O interface. The SPEs each have a 256KB local SRAM[9], and is connected to the EIB

[7] SSE did however share circuitry with the FPU, making it impossible to run SSE and FP instructions at the same time, but it eliminated the time spent switching between MMX and FPU operation.

[8] The various x86 extensions do get somewhat muddled together, since Intel and AMD have a tendency of adopting each others extensions, and adding on various functionalities which are then re-adopted by the other. Another example of this is SSE2, improved by AMD with the introduction of AMD64 by adding an additional eight registers, which was mimicked by Intel when they adopted AMD's 64-bit instruction set as EM64T in 2004.

[9] *Static Random Access Memory*, a memory technology that uses several transistors, typically six, to store one bit. It differs from the cheaper *Dynamic RAM* in that it is faster, and does not need periodic refreshing. However, it is much more expensive, since it requires significantly more chip space for the same storage.

through a DMA controller[10]. The Cell uses Rambus XDR DRAM, which delivers 12.8 GB/s per 32-bit memory channel for a total of 25.6 GB/s [IBMCELL1].

The design of the Cell opens for a variety of processing strategies. Six possible models are mentioned in [IBMCELL1], three of which will be mentioned here. In the *Function Offload Model*, the PPE is used much like a normal CPU, with different functions being offloaded from the PPE to different SPEs, allowing them to be executed in parallel. This model is similar to the one used by FPGA-based application acceleration. The *Computational Acceleration Model* is more centered on the SPEs, where parallelization techniques are used to partition work among the SPEs, and shared memory or message-passing techniques are used to coordinate execution between them, much like in a standard SMP architecture. The *Streaming Model* is somewhat similar to the previous, where the eight SPEs work together as a form of vector processor, each doing the same operation on different data in serial or parallel data streams.

Architectural details on the Cell can be found at [IBMCELL2], available at IBM's online documentation library. An overview, including information about the first commercial application of the Cell, in the Sony PlayStation 3, can be found at [WIKI05] and [WIKI06]. These articles also have many references to further sources of information.

[10] *Direct Memory Access*, a technique that uses an independent controller to reduce I/O overhead in the CPU, by directly mediating transfers between the local memory and other hardware subsystems.

3 Contemporary FPGA platforms

This section will do a short overview of some of the more popular platforms that employ FPGAs for computing and application acceleration. In the first two systems, the FPGA is integrated into the system itself, while the third is a more general description of FPGAs of the plug-in variety that can be used from many different systems.

A list of older systems can be found at [GUCC99], while a more comprehensive list that includes contemporary systems can be found at [FFAQ05].

3.1 Cray XD1

The Cray XD1, which is the focus of this text, is based around a relatively traditional setup, where you have several base chassis that can be interconnected in a variety of ways. Each chassis contains six compute blades, or nodes, and each compute blade hosts two AMD Opteron CPUs and a Xilinx FPGA, as well as its own memory and communication processors. Each node is an independent SMP computer in itself, having its own GNU/Linux OS image, and is connected to other nodes through the proprietary RapidArray interconnect.

The FPGAs, being attached to one particular node, can currently only be accessed by its host node. This creates a static assignment of two CPUs to one FPGA, which makes for a relatively cheap design, but could cause load imbalances between CPU execution and FPGA execution for some tasks. However, there is no particular ratio of CPUs to FPGAs that is optimal for all applications, so it is hard to draw any hard conclusions regarding the limitations of this design.

Further technical details on the Cray XD1 can be found in Appendix A.

3.2 SGI Altix

The SGI Altix series consists of several types of Intel Itanum 2-based computers, scaling from the mid-range server to the supercomputer segment. It is based around the concept of *bricks*, that are connected through the proprietary NUMAlink interconnect. The bricks are nodes that fill some function in the system, such as compute bricks, dedicated memory bricks and *Reconfigurable Application Specific Computing* (RASC) bricks. The latter house the FPGAs in the system.

Since the various bricks can be mixed and matched in a variety of configurations, with certain limitations depending on the particular server model, it is technically possible to balance the number of CPUs and FPGAs to suit a particular task. However, since the optimal configuration will vary between tasks, it would be hard or impossible to find a balance that would be generally optimal, so the value of this level of configurability is

limited unless the computer is to perform a small number of very well-defined tasks. Another advantage of this system is that each FPGA is not strictly bound to a set of CPUs, further increasing flexibility, but the looser connectivity also causes a higher latency for communication [SGIRASC].

3.3 PCI/PCI-X Expansion Cards

PCI and PCI-X-based expansion cards[11] that house FPGAs are available from a number of vendors, such as Nallatech[12], ClearSpeed[13] and Annapolis Micro[14]. These cards typically have one or more FPGAs, and could also have on-board RAM, while some cheaper cards use the host computer's memory exclusively. Many cards also have expansion slots, where D/A- and A/D-converters, memory expansion modules and other task-specific devices can be plugged in.

The big advantage of these solutions is that they can be plugged into virtually any computer, thus offering very flexible FPGA-based computing. They also come in a variety of configurations, with some cards having expansion facilities as mentioned above, avoiding the "one size fits all" problem. The big disadvantage is speed, since the communication bandwidth and latency of these cards are limited by the PCI bus[15], which is also shared with any other PCI expansion cards present in the computer. However, these limitations depend on the particular PCI bus implementation used, as well as the level of contention on the bus, and could very well be acceptable for all but the most demanding of tasks.

3.4 System Comparison

The most important differences between these three system variants are listed in Table 3.1. While the level of detail here is insufficient for an in-depth comparison, some general conclusions can be drawn. The Cray and SGI solutions both provide a FPGA solution with a fast but proprietary connection to the rest of the system, while the PCI expansion card solution is dependent on the PCI bus of the system, which is typically slower and has a higher latency. Cray provides a fixed 2:1 ratio of CPUs and FPGAs, while an arbitrary ratio can be configured with SGI as well as with the PCI cards, albeit a high number of FPGAs to CPUs in the latter case leads to increased contention on the PCI bus.

The task in hand decides which of these systems is the best suited, and with cost usually being an issue, the expansion card solution will often be sufficient. On the other hand, with HPC tasks that demand high scalability and bandwidth as well as low latency,

[11] PCI, or *Peripheral Component Interconnect*, is a computer bus standard for attaching peripheral devices, typically integrated circuits or expansion cards, to a computer motherboard.
[12] http://www.nallatech.com/
[13] http://www.clearspeed.com/
[14] http://www.annapmicro.com/
[15] 133 MB/s for traditional PCI, 533 MB/s for PCI 2.2, 1066 MB/s for PCI-X, 2133 MB/s for PCI-X 2.0.

solutions like the systems from SGI and Cray would come out on top. Picking one of these as an all-around winner in the HPC segment is not a trivial task however, as the differences in interconnects and base solutions make it hard or impossible to do an analytical analysis of the difference in performance between these two systems, and the results would likely vary significantly with the parallel granularity and bandwidth demands of the tasks used in such an analysis.

System	***Cray XD1***	***SGI Altix***	***PCI expansion card***
CPU	AMD Opteron	Intel Itanum 2	System dependent
Bus type	RapidArray: 2 GB/s, Direct Connect Architecture (FPGA at 1.6 GB/s)	NUMAlink: 12.8 GB/s, Ring Architecture (FPGA at 3.2 GB/s)	PCI Bus: System dependent speed (133-2133 MB/s), Shared Bus Architecture.
FPGA memory	4x QDR SRAM, 1.6 GB/s R/W per SRAM.	3x QDR SRAM, 1.6 GB/s R/W per SRAM.	Variable, often customizable.
CPU:FPGA ratio	2:1	Arbitrary	Arbitrary, with bus limitations.
Cost	Medium	High	Low-Medium

Table 3.1: FPGA system comparison

4 Endeavors in FPGA-accelerated software development

Using FPGAs as application accelerators in HPC environments is a relatively new idea, and the amount of work done on the area is not yet very extensive. Still, some promising results have been presented. This section will look at a selection of recent results in the field, beginning with the use of Smith-Waterson algorithm on a number of Nallatech FPGAs, where speedups around 200x were achieved. Then it will discuss results from two programs developed on the Cray XD1, namely a traffic simulation implementation and a BLAS implementation, both presented at SC|05 (SuperComputing 2005).

4.1 Smith-Waterman algorithm on Nallatech FPGA Systems

[XCBIO53] presents an implementation of the Smith-Waterman sequence alignment algorithm done on a Nallatech BenNUEY FPGA motherboard, with tests done both on a single Xilinx Virtex-II XC2V6000 FPGA as well as two FPGAs chained together. The FPGA implementations were compared to an implementation on a SunFire 280R, running two 1.05GHz UltraSPARC III processors with 8MB L2 cache and 8GB memory.

The Smith-Waterman algorithm is based on dynamic programming, and works on the basis of finding a value representing the closest "edit distance" between two strings, in this case representing chains of nucleotides or amino acids. It does this by applying a recursive condition where the similarity between two sequences are computed by taking the similarity of corresponding paired subsequences, and adding a penalty for mismatches occurring between them, where the penalty depends on whether an element has to be added, removed or changed to make them equal. A more verbose description is given in the article.

The implementation uses a number of processing elements (PEs), each hard-coded with parts of the query sequence, limiting the length of the query sequence by the number of PEs that can fit within a given FPGA multiplied by the number of FPGAs available. Proprietary tools from Nallatech were used to specify the network configuration of these elements, as well as bridging information between the different physical FPGA devices, after which the design was synthesized using the tools provided by Xilinx, and run against sequences from the GenBank database.

The results published indicated that the speedups obtained were significantly different between the different datasets, ranging from 2.2x (1 FPGA) and 2.9x (2 FPGAs) on the GBUNA data set, to 139.6x (1 FPGA) and 186.5x (2 FPGAs) on the GBPRI20 data set. The article claims that a speedup over 200x was achieved when three FPGAs were used, but details and results from these runs were not provided[16].

[16] It should be noted that one of the authors is an employee of Nallatech, so the objectivity of the article cannot be established. Certain assumptions and simplifications, like the hard-coded sequences, could have been done to increase the speedup at the expense of flexibility, but the article is still an interesting case.

4.2 Metropolitan Road Traffic Simulation on Cray XD1

[SC05311] presents a case study where a Cray XD1 supercomputer was used to run a road traffic simulation called TRANSIMS. The presentation focuses not only on the implementation on the FPGA, but also considers the software parts needed by the implementation, as well as the overhead implied in communication between the CPU and FPGA. These factors are then put together, amortizing the cost of communication against the pure FPGA speedup, resulting in a final speedup comparison with one CPU/FPGA pair compared to a sequential implementation running on only the CPU. A description of the algorithm and the implementation that is more verbose than the one in the paper presented at SC|05 is available at [TRANS05], and information from both these papers is combined here.

The TRANSSIMS algorithm, developed at Los Alamos National Laboratory, is a sophisticated iterative algorithm for road traffic simulation. Demographic information about the surrounding area is used to create a synthetic population of the city, including household sizes and the number of cars per household. Travel plans are then generated for each individual of the synthetic population for a 24-hour day, describing exactly where and when that person intends to go that day. This travel plan is then used to feed the road simulator algorithm.

The road system itself is presented as a cellular automata, where the lanes of the road are split into cells, each 7.5 meter long. Each cell can hold one car, which travels at a velocity set to 0-5 cells per iteration step. Roads with several lanes are represented as parallel sets of cells, which allow cars to change lanes based on a set of rules described by the algorithm, taking into account other cars and approaching intersections. The velocity of the car in each cell changes with the conditions of the neighboring cells, slowing down if there is an obstacle in a forward-bound cell (provided it either can't change lane or won't benefit from doing so), and accelerating or maintaining speed otherwise. A stochastic (random) chance of the car slowing down for no reason is also introduced, to better reflect the non-deterministic nature of drivers.

Two implementations of this algorithm on FPGAs are discussed. The simplest approach is to do a direct implementation of the algorithm, where each cell has a physical presence in the FPGA, but this approach requires a large number of FPGAs even for moderately sized cities. Using this approach, where all the calculations were performed on the FPGA more or less independent on the CPU, the actual speedup reached as high as 1175x. However, due to said limitations of available FPGA area, large-scale simulations using the direct approach do not scale well. The example used in the article, a model of the city of Portland, which has roughly two million inhabitants, would require 12400 FPGAs for a full implementation.

An alternative approach that is not as limited by available FPGA area is to use a computational engine that processes a stream of road data, updating the data one area at a time. Another consideration done in this design is which data should be processed by the FPGA, and which data should be processed by the CPU. The logic was significantly

simplified by doing intersection and multi-lane processing on the CPU, and limiting the FPGA to single-lane processing only, which fit well with the data used in the article, seeing as most of the roads were single-lane.

On the streaming implementation, speedups of to 126x were attainable if communication costs were disregarded, while the speedup dropped to 34x when the communication costs were included. Since this was for single-lane computation only, the total speedup was estimated to reach 5x if the entire data set could fit in the SRAM of the FPGA. Since the SRAM could only hold roughly two million road cells, the true speedup using this approach was only a rather abysmal 11%.

This approach used here is different from many of the other articles on the subject, which mainly focus on small computation-intensive parts of the code and compare the measured speed directly with software implementations of these parts alone, instead of looking at the speedup for the whole program. Naturally, the speedup obtained for the program as a whole is limited by Amdahl's Law[17], and this can therefore be considered a better metric for deciding whether the additional work involved in creating the FPGA part is reasonable. The lesson learned from this example is that a speedup of, say, 1000x on parts of the program does indeed sound impressive, but if these parts only account for 10% of the computation time, the speedup for the program as a whole will never exceed approximately 11%, and will likely be less when the communication overhead is taken into account.

4.3 High Performance Linear Algebra on Cray XD1

[SC05209] presents an implementation of parts of the Basic Linear Algebra Subprograms (BLAS) library using the FPGAs on the Cray XD1 supercomputer. BLAS is used in a wide range of software, and is a building block in many of the most important scientific libraries, such as LAPACK. It is also used by LINPACK and the popular LINPACK test, commonly used to rank the performance of supercomputers.

The implementation presented is a 64-bit double-precision matrix multiplier, which is able to scale over all the nodes in a Cray XD1 chassis, with the FPGAs communicating directly through RocketI/O links between the nodes. Sustainable performance on a single FPGA was measured to 2.06 GFLOPS, which is really not that impressive considering that the AMD processor can perform at 4.1 GFLOPS. The performance is mainly inhibited by the performance of the floating-point units, seeing as each FPU consumed almost 10% of the available FPGA area, and degraded the maximum clock speed from 200MHz to around 150MHz. However, according to the authors, the design should be able to scale to 12.4 GFLOPS using an entire chassis, and 148.3 GFLOPS using a

[17] Amdahl's Law is a formulation of the "law" of diminishing returns, stating that the total speedup for a program as a whole, caused by the speedup of a part of the program, can never exceed 1 / ((1-P) + (P/S)), where P is the proportion of the computation time affected by the speedup, and S is the speedup achieved. Note that Amdahl's Law does not consider the effects of problem size, which in some cases can be scaled up to leverage against this restriction (i.e., a larger problem size can lead to a larger P).

configuration of 12 chassis, so it is not unlikely that the solution scales better than a normal multiprocessor implementation of BLAS.

The article explores some particulars of the implementation, but does not consider how the CPUs could be utilized together with the FPGAs to increase performance, nor does it present any actual measurements on how well the design scales compared to a normal multiprocessor implementation. The first point could be intentional however, since the design presented allows the CPUs to perform other work while the BLAS operations take place, but the viability of such an approach depends on there being enough work for the CPUs to keep busy. If the FPGA implementation is unable to perform the operations faster than the CPU, and there is no other work that can be performed in parallel, the operations can just as well be performed on the CPU, saving the development efforts required to create the FPGA design.

II The Software Developer's Guide to the Cray XD1

This part will introduce the Cray XD1 seen from a software developer's point of view. Section 5 will first give an overview of the programming environment of the Cray XD1, followed by the usage of the on-board Field Programmable Gate Arrays (FPGAs), both from the command line and from applications. This part assumes some knowledge of the GNU/Linux command-line environment and the C programming language, and focuses mainly on the FPGA-related concepts of software development. Section 6 will then discuss higher-level distributed computing on the Cray XD1 platform, with focus on an OpenMP/MPI mixed-mode variant and how this can be combined with the FPGAs.

Part II does not contain any information about how the FPGA designs themselves are made; for this, see Part III.

5 Cray XD1 Program Development

Program development on the Cray XD1 is based around a standardized set of tools and libraries commonly available on the GNU/Linux platform, most of which should already be known to the readers, along with certain modified or proprietary additions particular to the Cray XD1 system. A throughout examination of all these tools and libraries is beyond the scope of this text, only the ones particular to the Cray XD1 being examined in detail.

5.1 The Programming Environment

The Cray XD1 programming environment is based around a set of *nodes*, each running one instance of the SuSE GNU/Linux operating system. Each XD1 chassis hosts six nodes, and several chassis can be connected together to increase the number of available nodes. The nodes are typically partitioned into *interactive nodes* and *compute nodes*, where the former are used to host interactive sessions and perform interactive tasks, such as compilation, debugging and task scheduling. These can be accessed remotely via **ssh**. The latter are usually off-limits for interactive use, and execute jobs that are dispatched through the scheduler.

The most important standard tools available for interactive work on the Cray XD1 are described in Section 5.1.1, and a list of the key libraries available can be found in Section 5.1.2.

Further details on the Cray XD1, including hardware configuration and details on operating system components, like the scheduler, can be found in Appendix A.

5.1.1 Standard Tools

The system comes installed with the standard GNU toolset for program development, from which the most important tools are listed below. These can be invoked from the command line in the same way as on other distros.

Text editors	`emacs, vim`
Compilers	`gcc, g++, g77`
Debuggers	`gdb`
Scripting	`perl, python`
Documentation	`groff, info/makeinfo`

Obviously, other tools could be installed and made available by the system administrator.

5.1.2 Key Libraries

The libraries included with the Cray XD1 that are relevant for HPC can be divided into two main areas: scientific and mathematical routines, and communication routines. In addition there are libraries for general operation, as well as proprietary libraries to access devices particular to the Cray XD1. The most important of these libraries are listed below. Optional libraries are denoted with an asterisk (*).

General	GNU C Library (glibc)
Mathematical	AMD Core Math Library (ACML) Scalable Linear Algebra Package (ScaLAPACK)
Communication	Message Passing Interface (MPI) ROMIO Aggregate Remote Memory Copy Interface (ARMCI) Global Arrays (GA) Generalized Portable SHMEM (GPSHMEM)
FPGA	Cray XD1 FPGA application acceleration processor interface library
Profiling	Performance Application Programming Interface (PAPI) * Cray Performance Analysis Tool (CrayPAT)

For a complete list of the available libraries, as well as include paths, a description of the various libraries and an explanation regarding the use of module files, see [S-2433] pages 10-26.

5.2 Using the FPGA Application Acceleration Processor

Each node in the Cray XD1 system houses two AMD Opteron CPUs as well as an optional reconfigurable Application Acceleration Processor (AAP) in the form of a Xilinx Vertex-II Pro FPGA. The rest of this discussion assumes that the FPGA is present and working. The technical details of the Cray XD1 can be found in Appendix A, while the general concept of using a FPGA as an AAP was discussed earlier in Section 2.4.

To use the FPGA, it must first be loaded with a FPGA design. The internals of such designs vary significantly, and can perform a virtually arbitrary set of functions, only limited by the size and performance of the FPGA itself. Because of the high complexity of making such designs, the most realistic option for the average software developer is to obtain designs through libraries that are made for the FPGA. These could either be general libraries developed by Hardware Intellectual Property (IP) companies, or libraries developed by in-house or consultant hardware developers in conjunction with the software developer. However, if the software developer also has skills in hardware development, it is of course possible to develop the two parts together. The hardware development process is described in Part III, and will not be discussed further in this part.

After a library has been obtained in one way or the other, there are two ways to access the FPGA in the Cray XD1. The first, described in detail in Section 5.3, involves using a command-line tool called **`fcu`**. The second, described in detail in Section 5.4, involves accessing it through a C application via the FPGA Application Programming Interface (API). Parts of the functionality of these two methods overlap, such as the ability to load, reset and unload the FPGA, as well as accessing limited status information. However, some of the functionality can only be found in one of the two. In particular, only the **`fcu`** tool can be used to prepare a raw binary file for the FPGA, as well as to generate the header files used for this preparation. Likewise, the memory and internals of the FPGA can only be manipulated through the FPGA API.

A detailed description of how to access the FPGA through these two methods will be given in the following sections. An illustration of the typical software development flow when using FPGAs, and the concepts involved to use the FPGA for a software designer, is given in Figure 5.1, adapted from [S-2433]. All the concepts within the border will be discussed in this part, while the hardware-specific part, grayed out in the illustration, will be discussed in Part III, Section 7.

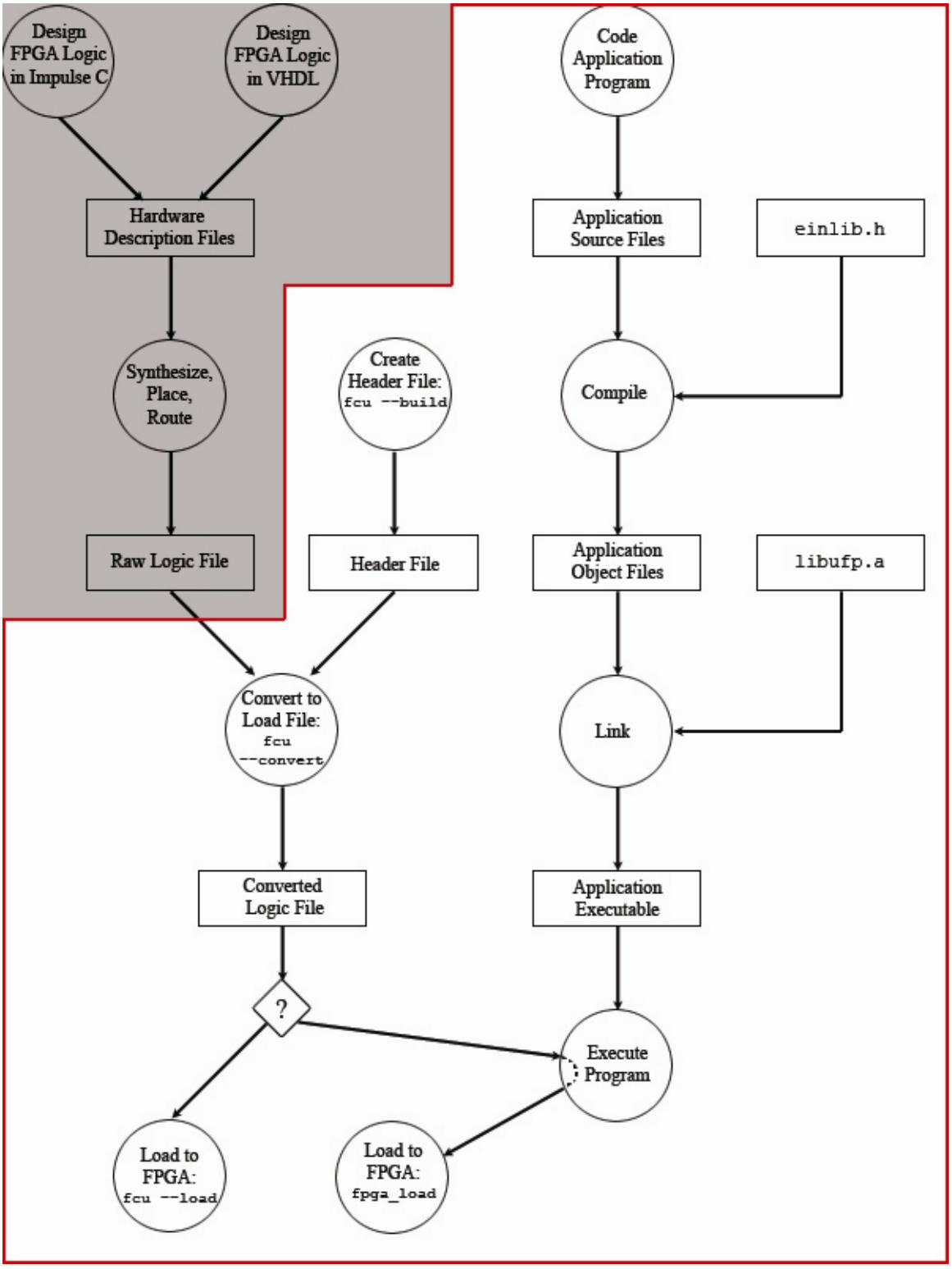

Figure 5.1: Typical FPGA software development workflow.

5.3 The FPGA Command-Line Interface

The Cray XD1 is equipped with a proprietary tool called **`fcu`**, which is used for accessing and manipulating the FPGA from the command-line interface. It is not as flexible as the C API (discussed later), and its most important use is to prepare raw binary files for use with the FPGA. This particular use is covered in Section 5.3.1. The tool also shares some functionality with the C API, described in Section 5.3.2. Table 5.1 shows a quick overview over the available commands and functionality. Additional information can be found by typing **`fcu –h`** or **`man fcu`** at the XD1 command prompt, as well as in [S-2433] and [S-6400].

Switch	Additional arguments	Description
-b | --build	[headerfile] [--partnum <part_number>] [--clock <clock_freq>]	Builds a header file, which can be used together with the binary file to create a loadable FPGA file.
-c | --convert	<rawfile> <headerfile> [loadfile]	Combines a raw binary file and a header file (created with -b) to create a loadable FPGA file.
-e | --exec		Enables execution on the FPGA, releasing it from a **reset** state.
-h | --help		Prints the help text.
-i | --info	<loadfile>	Displays the header information from the specified loadfile.
-l | --load	<loadfile>	Downloads the specified loadfile to the FPGA, overwriting any previous content.
-r | --reset		Resets the FPGA, placing it in a **reset** state.
-s | --status		Prints status information from the FPGA.
-u | --unload		Erases any loaded logic from the FPGA.
-V | --version		Displays the version of the **`fcu`** tool.

Table 5.1: `fcu` **switches and arguments.**

5.3.1 Preparing binary files for use with the FPGA

To use the raw binary file generated by the designer tools with the FPGA, it has to be appended with a header file which tells the system the clock speed the FPGA operates at, as well as the part number of the FPGA. Certain modifications are also done to the byte structure of the binary file. The **`fcu`** tool can both build the header file, and perform this conversion.

To make the header file, use the following command at the command prompt:

```
> fcu --build [headerfile] [--partnum part-number]
   [--clock clock-freq]
```

headerfile defaults to `ufphdr` if it is not specified. If **`--partnum`** or **`--clock`** is omitted, the program will prompt for these values. If *part-number* is not known, it can be found with the command:

```
> lsnode –-verbose | grep "App Accelerator"
```

This will display a value on the form *87-nnnn-nn* or *90-nnnn-nn*. In the latter case, the value can be used directly, but in the former case the corresponding *90*-series number should be used instead. In particular, part number *87-0003-09* should use *90-0003-05* while *87-0003-11* should use *90-0003-08*. See [S-6400] page 11 for full details.

clock-freq defines the clock frequency of the FPGA in MHz, and is a value from 63 to 199, inclusive. The value to use here should be specified by the FPGA design. Note that if the SRAM core is used in the design, the value is further restricted to be from 130 to 199, inclusive, but this is not enforced by the **`fcu`** tool.

The header file generated is in plain ASCII format, and should resemble the following:

```
Cray Part Number   : 90-0003-08;
FPGA Frequency MHz : 199;
```

After the header file is generated, the following command will convert the binary file to the proper format:

```
> fcu --convert rawfile headerfile [loadfile]
```

rawfile is the name if the raw binary file, *headerfile* is the name of the header file generated earlier, and *loadfile* is the name given to the output file. If no *loadfile* is specified, it defaults to the full name of *rawfile* appended with the extension *.ufp*.

This will append the header and perform the necessary modifications to the logic file, after which it will be ready to be loaded to the FPGA.

5.3.2 Using the FPGA

Most of the other functions in the **`fcu`** tool are replicated in the C API, and due to its limited nature it is unlikely that they will be frequently used. This section will therefore just list a quick overview of the important ones, with the related function in the C API listed immediately after the command description.

`fcu --load` *loadfile* will load the prepared logic file *loadfile* to the FPGA, reset the logic, and finally release it from reset. This command could be used to load the

FPGA before the actual application is started, for instance via a Bash script, which could be useful if the same program is run a number of times. (**`fpga_load`**)

`fcu --unload` erases any logic programming from the FPGA. This is usually not needed, as an **`fcu --load`** or equivalent call from the C API will overwrite the information on the FPGA regardless of any currently loaded programs, but it can be useful if the secrecy or security of the loaded programming is an issue. (**`fpga_unload`**)

`fcu --reset` places the FPGA in a reset state by asserting the `user_reset_n` signal from the RT core. (**`fpga_reset`**)

`fcu --exec` releases the FPGA from a reset state by de-asserting the `user_reset_n` signal. (**`fpga_start`**)

`fcu --status` prints the value of the host latch register in the RT core as a decimal integer. A code of 255 indicates that the FPGA is not loaded. (**`fpga_status`**)

5.4 The FPGA Application Programming Interface

All application-initiated use and interaction with the FPGA happens through the FPGA Application Programming Interface (API). This section will introduce the API library, and do a throughout examination of the various functions provided by it. At the time of this writing, only a C API is available from Cray, so the discussion will be limited to using the FPGA from C.

As for languages other than C, normal rules apply for invoking the C API indirectly, but the exact approach is compiler-specific. In particular, FORTRAN follows the normal conventions for using C libraries, as described in [PGI60UG], but is outside the scope of this text and will not be explored further.

5.4.1 Library Files

The two files needed for accessing the FPGA from C programs are the include file, `/usr/local/include/einlib.h`, which is the header file used by C, as well as the object library, `/usr/local/lib64/libufp.a`, which is used by the linker. Using the header file is done through a straight-forward `#include “einlib.h”` compiler directive, while the object library is given as input to the linker when the program is compiled.

For example, the following command invokes the gcc compiler to compile a C program with the FPGA enabled:

```
gcc –I/usr/local/include –m64 filename.c -L/usr/local/lib64
 –lufp –o filename
```

5.4.2 Using the FPGA

The FPGA can be invoked from the C program using the API in much the same way as one would access any other device. There is no implicit synchronization between the CPU and the FPGA, and the communication is done through memory-mapped I/O. The API provides a variety of functions to manage the FPGA, as well as several communication modes including single transfers, memory-mapped transfers and FPGA-initiated transfers using a FPGA transfer memory region. Table 5.2 provides a short overview of the various API calls with a short description, while a complete reference table with function signature, arguments and return values is available in Appendix B. The various functions and communication varieties will be explored further in the sections below.

API call	Description
`fpga_open`	Opens a file descriptor used to communicate with the FPGA.
`fpga_load`	Loads the FPGA with a loadfile.
`fpga_reset`	Places the FPGA in the **reset** state.
`fpga_start`	Releases the FPGA from the **reset** state.
`fpga_status`	Retrieves a status value from the host latch register in the RA core.
`fpga_is_loaded`	Queries whether the FPGA is loaded or not.
`fpga_unload`	Erases any logic programming from the FPGA.
`fpga_close`	Closes the FPGA file descriptor.
`fpga_memmap`	Maps a region of the FPGA address space to the application address space.
`fpga_mem_sync`	Flushes all outstanding memory transactions from a particular memory-mapped area.
`fpga_wrt_appif_val`	Writes a single value to the FPGA.
`fpga_rd_appif_val`	Reads a single value from the FPGA.
`fpga_set_ftrmem`	Sets up a FPGA transfer region used for FPGA-initiated reads and writes.

Table 5.2: Overview of API calls in `einlib.h`

The generic example code used throughout the following sections will assume that certain values and initializations present in sections preceding them have been run. For instance, the code used to open and load the FPGA will not be repeated in later sections.

Template C file is found at: `/opt/ufpapps/vhdl_template/src/template.c`

5.4.2.1 Opening the FPGA

Before the FPGA can be used by the application, it has to be prepared for access with the **`fpga_open`** function, detailed in Table 5.3. This opens the FPGA as a file descriptor, which is then used by all other FPGA functions to communicate with the device. No actual communication happens with the FPGA at this point however; it only affects the application side.

`fpga_open` code example:

```
err_e err; /* Used to hold the error value from the API */
char * fpga_path = "/dev/ufp0";  /* FPGA filesys. handle */
int fpga_fd=0;                   /* FPGA file descriptor */

fpga_fd = fpga_open(fpga_path,     /* File system handle */
                    O_RDWR|O_SYNC, /* open flags */
                    &err);         /* Returned error */

if (err != NOERR) {
     /* Error handling */
}

/* FPGA is now opened as the file descriptor fpga_fd. */
```

Variable	Type	In/Out	Description
fpga_path	char *	in	The absolute path to the FPGA character device file; typically "`/dev/ufp0`".
flags	int	in	A bitwise OR of the appropriate masks used by the `open` system call; typically `O_RDWR|O_SYNC`. See [OGOPEN] and **`man open`** for all possible options.
err	err_e *	out	Returned error code from the operation.
fpga_fd	int	return	A file descriptor of the opened FPGA.

Table 5.3: Arguments and return value for `fpga_open`

5.4.2.2 Loading the FPGA

After a file descriptor to the FPGA has been opened, **`fpga_load`** can be used to load the FPGA with a logic file. Note that binary files from FPGA tools have to be prepared with the **`fcu`** tool before it can be loaded to the FPGA; see Section 5.3.1 for this particular use of the tool. This step can also be skipped if the FPGA was loaded using **`fcu --load`** before the program is started, as described in Section 5.3.2. After the load process completes, the FPGA logic is automatically reset and released, after which it is ready for use. Table 5.4 details the arguments of this function.

`fpga_load` code example:

```
char * loadfile = "loadfile.ufp";  /* Path to the file to
                                    * load the FPGA with */

int num_bytes = fpga_load(fpga_fd,  /* File descriptor */
                          loadfile, /* Loadfile path */
                          &err);    /* Returned error */

if(num_bytes == -1 || err != NOERR) {/* Error handling */}

/* The FPGA is now loaded and ready. */
```

Variable	Type	In/Out	Description
fpga_fd	int	in	The FPGA file descriptor.
loadfile	char *	in	The path to the FPGA logic file that is to be loaded.
err	err_e *	out	Returned error code from the operation.
num_bytes	int	return	The number of bytes written to the FPGA device, or -1 on failure.

Table 5.4: Arguments and return value for `fpga_load`

5.4.2.3 Checking status and programming state

The API provides two basic functions to check the state of the FPGA. The first of these, **`fpga_status`**, detailed in Table 5.5, retrieves a status value from the FPGA, namely the value of the host latch register (0x0C) in the RapidArray Core. This is an 8-bit integer, where a value of 255 (all bits set) means that the FPGA is not loaded, while the nominal value is 0 (all bits cleared) if the FPGA is loaded. Values other than these indicate an error condition in a loaded FPGA; in particular, bit 0 set indicates a Host Bus Parity Error (HOST_PERR), bit 1 set indicates a RT Rx Bus Parity Error (RT_PERR) and bit 2 set indicates a RT Rx Bus Unknown Command (RT_UNKN_CMD). Bits 3 through 7 are not used. See [S-6411] for further details.

The second function, **`fpga_is_loaded`**, detailed in Table 5.6, simply queries whether the FPGA is loaded or not, returning a value of 1 if it is loaded and 0 otherwise. It has a subset of the functionality of **`fpga_status`**, and can therefore be considered a convenience function.

WARNING: While the official documentation and man file for **`fpga_is_loaded`** claim that it returns 1 if loaded and 0 otherwise, this is confirmed false in release 1.2, which was used when this text was written. The bug was confirmed by Cray [S-2455 ref.

3760], and allegedly resolved in release 1.3, but this upgrade was not installed by the time this text was finalized. This text assumes that the correct output is given.

`fpga_status` and `fpga_is_loaded` code example:

```
int status = fpga_status(fpga_fd, /* File descriptor */
                         &err);   /* Returned error */

int loaded = fpga_is_loaded(fpga_fd, /* File descriptor */
                            &err);   /* Returned error */

printf("FPGA status: 0x%02X, FPGA loaded: %d\n",
     status, loaded);

/* status and loaded should be set to -1 in case of error,
 * so testing err is usually unnecessary. */
```

Variable	Type	In/Out	Description
fpga_fd	int	in	The FPGA file descriptor.
err	err_e *	out	Returned error code from the operation.
status	int	return	The returned status code, or -1 on failure.

Table 5.5: Arguments and return value for `fpga_status`

Variable	Type	In/Out	Description
fpga_fd	int	in	The FPGA file descriptor.
err	err_e *	out	Returned error code from the operation.
loaded	int	return	1 if the FPGA is loaded, 0 otherwise.

Table 5.6: Arguments and return value for `fpga_is_loaded`

5.4.2.4 Resetting and Releasing the FPGA

The API provides the two functions **`fpga_reset`** and **`fpga_start`**, detailed in Table 5.7 and Table 5.8, to reset the FPGA circuitry to its initial state. **`fpga_reset`** places the application logic into a reset state by asserting the `user_reset_n` signal output by the RT Core, while **`fpga_start`** takes the logic out of the reset state by de-asserting this signal. Calling **`fpga_start`** when the FPGA is not in the reset state has no effect.

The FPGA is automatically reset when **`fpga_load`** is called, but if the FPGA was loaded outside the currently executing program, either by using **`fcu --load`** or from

earlier programs, resetting the FPGA is recommended in order to put it into a known initial state.

WARNING: In the current release, **`fpga_reset`** should be used with caution. If any part of the program attempts to access the FPGA through the functions discussed in Section 5.4.2.5 while it is in a reset state, this will cause a FPGA timeout, bringing down the node and forcing a reboot. While the monitoring systems should bring the node back up within 5-10 minutes, any jobs running on the node will be lost, and there is always a potential for data loss. This is particularly something to watch out for with race conditions in parallel software implementations.

`fpga_reset` and `fpga_start` code example:

```
fpga_reset(fpga_fd, &err);

if(err != NOERR) {
     /* Error handling */
}

/* The FPGA is now in the reset state. */

fpga_start(fpga_fd, &err);

if(err != NOERR) {
     /* Error handling */
}

/* The FPGA is now released from the reset state. */
```

Variable	Type	In/Out	Description
fpga_fd	int	in	The FPGA file descriptor.
err	err_e *	out	Returned error code from the operation.
status	int	return	0 on success, or -1 on failure.

Table 5.7: Arguments and return value for `fpga_reset`

Variable	Type	In/Out	Description
fpga_fd	int	in	The FPGA file descriptor.
err	err_e *	out	Returned error code from the operation.
status	int	return	0 on success, or -1 on failure.

Table 5.8: Arguments and return value for `fpga_start`

5.4.2.5 Accessing the FPGA

There are three alterative methods to communicate with the internal memory of the FPGA. Note that the addressing used by all variants depends on the design running on the FPGA, and do not generally refer to any particular register or internal FPGA memory. Also, the first two are used only during CPU-initiated communication, while the third is used only by FPGA-initiated communication. The first variant involves using the functions `fpga_rd_appif_val` and `fpga_wrt_appif_val`, which will read or write a 64-bit value from or to the FPGA. The second is mapping the FPGA memory to a local memory area with `fpga_memmap`, and perform all reads and writes to this area. The third is creating a local memory area with `fpga_set_ftrmem` that the FPGA can read from and write to directly.

`fpga_rd_appif_val` and `fpga_wrt_appif_val`, shown in Table 5.9 and Table 5.10, are typically used at the start of a program to initialize various values in the FPGA. An important example of this is to give the FPGA a pointer to the shared memory buffer in the SMP DRAM created by `fpga_set_ftrmem`, to allow FPGA-initiated reads and writes. Another important use is for reads and writes that for some reason have to be performed in a certain sequence, as `fpga_memmap` uses write combining to increase performance.

`fpga_rd_appif_val` and `fpga_wrt_appif_val` code example:

```
u_64 wrt_val, rd_val, reg_offset, reg_address;

wrt_val = 31337;
reg_offset  = 1024*1024*64;  /* Start at 64M */
reg_address = reg_offset + 0x18UL;

fpga_wrt_appif_val( fpga_fd, /* File descriptor */
               wrt_val,      /* Value written to FPGA */
               reg_address,  /* Write address */
               0,            /* Don't perform conversion */
               &err);        /* Returned error */

/* Error checking goes here... */

fpga_rd_appif_val(  fpga_fd,  /* File descriptor */
               &rd_val,       /* Value read from FPGA */
               reg_address,   /* Read address */
               &err);         /* Returned error */

/* Error checking goes here... */

if(wrt_val != rd_val) { /* Raise error */ }
```

Variable	Type	In/Out	Description
fpga_fd	int	in	The FPGA file descriptor.
value	unsigned long	in	The 64-bit value to be written to the FPGA.
offset	unsigned long	in	The byte offset in the FPGA address space where the value is to be written.
type	unsigned long	in	0 if the value is to be written as it is; 1 if the value is a user-space virtual memory address in a FPGA transfer region (see the associated **man** page for details).
err	err_e *	out	Returned error code from the operation.
status	int	return	0 on success, or -1 on failure.

Table 5.9: Arguments and return value for `fpga_wrt_appif_val`

Variable	Type	In/Out	Description
fpga_fd	int	in	The FPGA file descriptor.
value	unsigned long *	out	The 64-bit value read from the FPGA, returned by the function.
offset	unsigned long	in	The byte offset in the FPGA address space where the value is to be read.
err	err_e *	out	Returned error code from the operation.
status	int	return	0 on success, or -1 on failure.

Table 5.10: Arguments and return value for `fpga_rd_appif_val`

`fpga_memmap`, shown in Table 5.11, is as mentioned before used to map a region of the FPGA address space into the application address space. The program can then read and write to this memory area using normal pointers, much like manipulating local memory. It is generally faster than using the single write/read functions, due to its use of the Opteron write combining features, and should therefore be used whenever possible. Note that while write bursts can be performed using this technique, read bursts cannot. It is therefore usually preferable that the data is written to the FPGA by the CPU, with the results being written back to the CPU by the FPGA instead of being read by the CPU. This is possible with the use of `fpga_set_ftrmem`, discussed below.

`fpga_mem_sync`, shown in Table 5.12, is an auxiliary function that can be used on memory areas mapped with `fpga_memmap` whenever it is necessary to maintain sequence of writes to the FPGA. It works like a barrier, ensuring that all outstanding memory writes have been completed before any further accesses are performed. It is usually preferred to avoid using the single write/read functions except for in some cases, as discussed above, and instead invoke this function in order to force sequential writes whenever this is actually required.

`fpga_memmap` and `fpga_mem_sync` code example:

```
size_t len;
off_t off;
int prot, flags;
u_64 * fpga_base;
u_64 ptr;

len = 1024*1024*16; /* 16MB memory map */
off = 0;            /* Start at offset 0 in the FPGA's
                     * address space */

prot  = PROT_READ|PROT_WRITE; /* Allow read/write */
flags = MAP_SHARED;           /* Shared mapping */

/* Call memmap, and get a pointer to the mapped memory */
fpga_base = (u_64 *)
     fpga_memmap(fpga_fd,     /* File descriptor */
                 len,         /* Size in bytes */
                 prot,        /* Memory protection flags */
                 flags,       /* Memory sharing flags */
                 off,         /* Memory offset */
                 &err);       /* Returned error */

if(fpga_base == NULL) {
     /* Memory allocation failed - Handle error */
}

ptr = 0; /* Start writing at first byte in memory map */

*(fpga_base+ptr) = 0xABCDEFDADBEDDEDA;

/* Synchronize memory (guarantees that the previous write
 * will finish before the next one is posted). Note that
 * in many cases this is unnecessary, and should only be
 * done if the FPGA depends on it. */
fpga_mem_sync(fpga_fd, &err);

/* Error checking goes here... */

ptr += sizeof(u_64);

*(fpga_base+ptr) = 0xBADBADBABEFEDCBA;
```

Variable	Type	In/Out	Description
fpga_fd	int	in	The FPGA file descriptor.
length	size_t	in	The number of bytes to be mapped.
protect	int	in	A bitwise OR of the memory protection flags specified by the `mmap` system call; typically `PROT_READ|PROT_WRITE`. [OGMMAP]
flags	int	in	A bitwise OR of the mapping options specified by the `mmap` system call; typically either `MAP_SHARED` or `MAP_PRIVATE` (only one of these can be specified at any time). [OGMMAP]
offset	off_t	in	The byte offset in the FPGA address space at which this mapped region begins.
err	err_e *	out	Returned error code from the operation.
fpga_base	void *	return	A pointer to the mapped area in the application memory space, or `NULL` on failure.

Table 5.11: Arguments and return value for `fpga_memmap`

Variable	Type	In/Out	Description
fpga_fd	int	in	The FPGA file descriptor.
err	err_e *	out	Returned error code from the operation.
status	in	return	0 on success, or -1 on failure.

Table 5.12: Arguments and return value for `fpga_mem_sync`

`fpga_set_ftrmem` enables the FPGA to invoke communication between the SMP and itself. When the CPU calls this function, it sets aside an area of memory where the FPGA can read or write as it pleases. However, as mentioned earlier, the pointer to this memory area first needs to be passed to the FPGA either through the `fpga_wrt_appif_val` function or through a write to a memory map created by `fpga_memmap`.

FPGA-initiated communication is in many cases vital to exploit the available performance of the system, seeing as the CPU cannot have outstanding reads from the FPGA, and has to wait for each one. The FPGA is however free to both post writes and have several outstanding reads (up to 32), which increases performance substantially, due to the internal “tag” system described in Section 7.4.1. Of course, if the result set is relatively small, with the amount of data transferred from the FPGA to the CPU being negligible, the added complexity may not be worthwhile.

`fpga_set_ftrmem` code example:

```
#define SMP_MEM_PTR_REG (1024*1024*64) + 0x20UL
     /* A register in the user logic holding the memory
      * pointer (user defined). */

...

volatile u_64 * ftr_mem;  /* Can be changed externally */
int order;

order = 9;     /* 2MB memory area */

/* Allocate FPGA transfer region */
ftr_mem = (u_64 *)
     fpga_set_ftrmem(fpga_fd, /* File descriptor */
                     order,   /* Size of FTR */
                     &err);   /* Returned error */

/* Test for error */
if(ftr_mem == NULL) {
     /* Memory allocation failed - Handle error */
}

/* Write memory pointer to FPGA */
fpga_wrt_appif_val( fpga_fd,       /* File descriptor */
                    SMP_MEM_PTR_REG, /* FPGA mem ptr reg */
                    (u_64)ftr_mem, /* Memory pointer */
                    1,             /* Perform conversion */
                    &err);         /* Returned error */

/* Error checking goes here... */
```

Variable	Type	In/Out	Description
fpga_fd	int	in	The FPGA file descriptor.
order	unsigned long	in	$\log_2$ of the number of 4K memory pages to be allocated; that is, $4K*2^{order}$ bytes. Valid values are 0 (4K) through 9 (2MB).
err	err_e *	out	Returned error code from the operation.
ftr_mem	void *	return	A pointer to the allocated FPGA transfer region, or `NULL` on failure.

Table 5.13: Arguments and return value for `fpga_set_ftrmem`

5.4.2.6 Erasing the FPGA

The **`fpga_unload`** function, detailed in Table 5.14, can be used to erase any loaded programming from the FPGA. This is usually not required, as **`fpga_load`** will overwrite the information on the FPGA regardless of any currently loaded programs, but it can be useful if the secrecy or security of the loaded programming is an issue.

`fpga_unload` code example:

```
fpga_unload(fpga_fd, /* File descriptor */
            &err);   /* Returned error */

if(err != NOERR) { /* Error handling */ }

/* The FPGA is now unloaded. */
```

Variable	Type	In/Out	Description
fpga_fd	int	in	The FPGA file descriptor.
err	err_e *	out	Returned error code from the operation.
status	int	return	0 on success, or -1 on failure.

Table 5.14: Arguments and return value for `fpga_unload`

5.4.2.7 Closing the FPGA

The **`fpga_close`** function, detailed in Table 5.15, is used to close the FPGA file descriptor, and is typically invoked before a program terminates. Not that the file descriptor becomes invalid after this function is called, and any further calls that attempts to use this file descriptor will return an *"Invalid API operation requested"* error. The function also clears all associations to memory areas established by **`fpga_memmap`**.

`fpga_close` code example:

```
fpga_close(fpga_fd, /* File descriptor */
           &err);   /* Returned error */

if(err != NOERR) {
     /* Error handling */
}

/* The FPGA file descriptor is now closed. */
```

Variable	Type	In/Out	Description
fpga_fd	int	in	The FPGA file descriptor.
err	int *	out	Returned error code from the operation.
status	int	return	0 on success, or -1 on failure.

Table 5.15: Arguments and return value for `fpga_close`

5.4.3 Security and stability considerations

Obviously, there is a serious consideration to do regarding the security and stability of the Cray XD1 platform. Since any user with access to the system can easily bring down all nodes that allow using the FPGA, employing it as a critical system can hardly be endorsed. While writing this text, the node used for testing was crashed no less than thirty-five times, with few of these being intentional, and many having a less than obvious cause. In some cases, the reason for the crash could not be determined.

Another potential problem is with memory protection when doing data accesses from the FPGA to the SMP node. According to a Cray representative at a product presentation during the spring of 2005, the FPGA has full and complete access to read from and write to the entire memory of the host SMP node. Obviously, this is not very viable containment-wise, as a program running on the FPGA could read kernel-level data and even change kernel-level data structures, which for instance could be used to grant the user root-level access, or failing that, perform any needed operations directly by manipulating the memory structures.

During one of the tests, a faulty signal association caused the FPGA logic to accidentally zero out the entire main memory of the node instead of just the FPGA transfer region, as was intended, and this obviously brought the node down fairly fast. Unfortunately, due to time constraints and the various problems summed up in Appendix C, it was not possible to do a comprehensive overview of the problems with memory protection by the time this text was finalized, but it should be fairly easy to create a program that allows someone, even a non-privileged user, to read from, display, and alter any memory location on the SMP node. Even assumed that only trusted users are granted access to the system, much more care than usual needs to be taken to avoid faulty memory writes, increasing the burden on the designer.

6 Using the system with MPI and OpenMP

Typically, the Cray XD1 will be configured with a number of chassis with six nodes each, where each node has two single-core or dual-core AMD Opteron CPUs as well as a FPGA. The problem is therefore not as simple as dividing a workload between one CPU and one FPGA. The question of how to divide the work between the nodes in a way that avoids having the increase in computation capacity being lost in the communication overhead is also important. To make things even worse, there are two to four CPU cores for each FPGA, meaning that a form of arbitration mechanism is needed. As the final straw on the camel's back, the FPGAs in a chassis can also communicate directly between each other over the RocketI/O interface[18].

This section will explore some possible ways to cope with this complexity, by using a combination of the standardized libraries OpenMP and MPI, for shared-memory computing and distributed-memory computing respectively. The concept is illustrated in Figure 6.1.

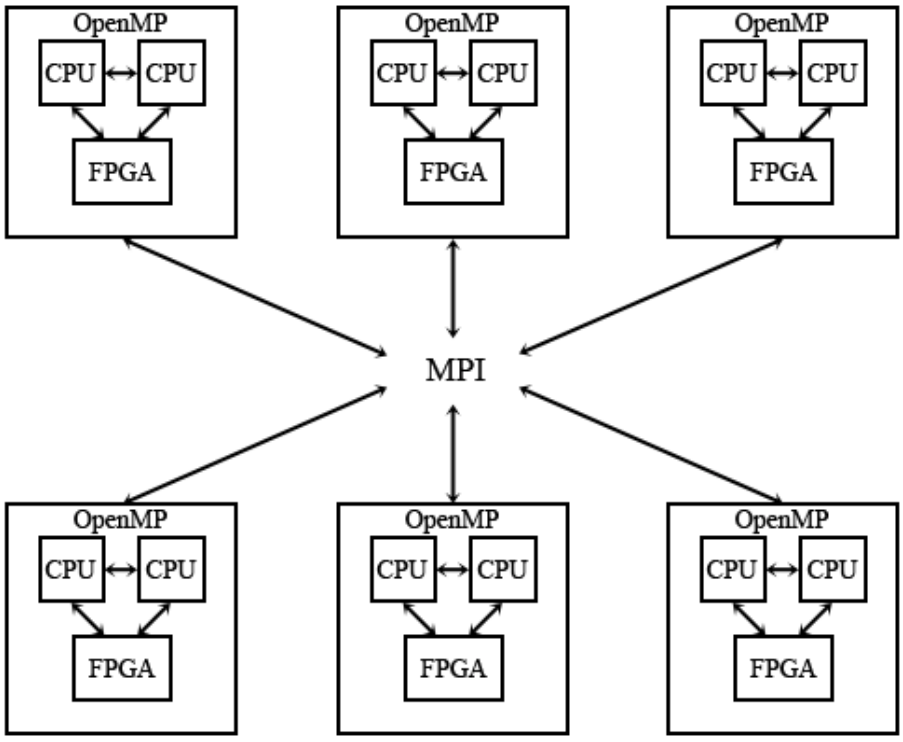

Figure 6.1: Mixed-mode MPI and OpenMP.

[18] Note that this way of communication was not explored while writing this text, since it is not documented by Cray. The links are illustrated in some of the figures in the official documentation, and is mentioned in [SC05209] as a part of the BLAS implementation discussed in Section 4.3, but is not mentioned elsewhere in the documentation made available at the time of this writing.

Some background knowledge of MPI and OpenMP is needed for this section, but good online tutorials are available for the novice reader, such as [TUTOMP] for OpenMP and [TUTMPI] for MPI.

Note that this section also touches on some concepts on the hardware level while discussing different ways of accessing the FPGA from several parallel threads.

To run MPI and OpenMP programs on the system, knowledge of the scheduler is also required. Among other things, this offers the ability to reserve the nodes on the system, so that conflicts regarding use of the FPGAs can be avoided. The scheduler will not be covered here, and the reader is referred to the associated `man` page (`man pbs`).

6.1 Choice of Strategy

Each SMP node consists of two CPUs that share the same memory. Using a library specialized for shared-memory architecture on each node is therefore natural. The choice here usually falls on OpenMP, due to its status as the *de facto* standard, and this is what will be explored here. However, other similar libraries such as SHMEM are also supported by the Cray XD1, and could be used in place of OpenMP.

Similarly, the various SMP nodes are organized in a distributed-memory fashion, and therefore cannot directly access each other's memory. A form of message-passing library is natural in this setting, whereof MPI, like OpenMP, is the *de facto* standard, and therefore explored here.

It is also possible, and not uncommon, to use MPI even for intra-node communication. This has the advantage of creating a more uniform model for splitting data between the CPUs, but has the distinct disadvantage of not exploiting the lower memory latency between CPUs on the same node. Still, on a properly optimized MPI implementation, intra-node latency need not be significantly higher than with OpenMP. On the Cray XD1 there is another advantage of using OpenMP for intra-node communication, namely that the arbitration towards the FPGA is more explicit. That is, all the nodes that communicate through OpenMP also share the same FPGA, so it is conceptually easier to perform any required arbitration.

[SMITH01] discusses advantages and disadvantages with mixed-mode MPI/OpenMP, including problems inherent with each of the paradigms and how they can be combined to mitigate against these problems.

It should be pointed out that while communication with OpenMP is relatively simple, due to its mostly implicit nature and use of compiler directives to more or less automatically parallelize code, MPI is far more explicit and typically requires a much larger amount of logic to work. Developing a mixed-mode system incurs the cost of both implementations, and parallelizing complex systems in this way is therefore a very large and complex task.

For less demanding applications, sticking with either OpenMP or MPI is therefore recommended.

6.2 The Load Balancing Problem

For all parallel program implementations, the problem of load balancing has always been one of the major points to consider. In short, the problem is this: how should the computations be split up between the computation devices to achieve the highest possible utilization at each unit, at the lowest possible communication cost?

The answer is highly dependent on both the problem and the platform. If the problem has high granularity (i.e., the various parts of the computation can to a high degree be performed independent on the other parts of the computation), the program can be evenly split up with relative ease, with little regard of the latency inherent in the underlying platform. On the other hand, a problem with low granularity cannot easily be split up on a platform with relatively high latency between the devices without introducing devastating communication overheads, which typically is the case with distributed-memory systems, while it is still possible for platforms with low latency, typical for shared-memory systems.

Another factor in this equation is implementing data-parallel versus control-parallel programs. In the former, all the computation devices perform the same operations, only on different sets of data. In the latter, the computation devices perform different operations. For instance, in a mixed-mode implementation, one thread on each node could be occupied with I/O and inter-node communication while the other nodes perform the required computations. Typically, data-parallel programs are both the most common and the easiest on which to attain a similar load on the various devices, but control-parallel systems are not uncommon, and variants of this approach are often seen in coprocessors such as DMA controllers.

As mentioned before, the Cray XD1 uses a combination of shared memory and distributed memory, so the best solution boils down to attempting to fit computation parts with low granularity and high interdependence on the same SMP node, where they are computed using OpenMP, while larger and less interdependent parts are distributed between the nodes, where communication happens through MPI. It is however highly unlikely that a cut-and-dried solution to all given problems exist, so there will usually have to be done compromises between parallelization and communication overhead. In some cases, the overhead could get so large that it is useless to parallelize the program further, while in other cases large overhead could be due to a poor implementation. Determining what is viable to parallelize and what is not is however a matter of experience, and is not easily condensed to a few pages of text.

6.3 Inter-Node Communication

The first obstacle to creating a parallel program is to find a high-level way of partitioning the problem, in a way that allows each node to perform a part of the computation that is as independent of the other parts as possible. This is not an easy task by any metric, and the level of interdependence will vary greatly from problem to problem. Unfortunately, so will the solution methodology.

Typical candidates for parallelization include iterative approximate algorithms for a variety of physical problems, such as weather forecasting and climate models. Typically, such candidates have a certain data locality, so that most of the calculations depend on data values that are "close" to each other in some respect. Parallelizing the algorithm is thus a matter of figuring out what parts of the data are computationally "close", and to partition the problem into parts where as little as possible of this data is located in other parts. Nevertheless, there is virtually always some need for communication between the various parts, to retrieve and update remote data, so the goal is usually to minimize this communication.

When a suitable partition is found, the data that needs to be passed around between the parts has to be charted, and the necessary communication must then be coded using the MPI primitives (such as `MPI_Send/MPI_Recv`). This process interacts with the work of creating the intra-node communication described below, as the rules for which thread is to perform the inter-node communication need to be decided. One solution is to make the master thread take care of all communication, while another is to distribute the MPI calls between the threads. However, care should be paid to the fact that not all MPI implementations are thread-safe, so if a solution where the various threads on a node collaborate on inter-node communication is desirable, the multithreading support for the MPI implementation has to be checked. OpenMP `critical` areas can however be used to mitigate against non-supporting MPI implementations in this case.

6.4 Intra-Node Communication

As soon as a problem has been split into relatively loosely connected parts that can be distributed between the various nodes, it is time to further divide the problem between the computation devices within the nodes themselves. In a typical SMP system, there are a number of identical processors (thus the name *Symmetric MultiProcessing*), and the intra-node partitioning could be as simple as using the `parallel for` compiler directive to create data-parallel loops, and possibly the `parallel sections` compiler directive to create control-parallel sections. These could also be made parts of a larger parallel region, to avoid superfluous fork/join operations.

However, the Cray XD1 is again somewhat more complex than the typical system. In addition to balancing a problem over the CPUs in each node, there is also the issue of what has been the center of attention so far in this text, namely the FPGA Application

Acceleration Processor. What is possibly the most challenging problem is moving the parts most suited for the FPGA to it, while still keeping the CPUs busy with useful work. It is far from sure that this will always be possible. In fact, it is quite likely that the end result will be either a design that puts a high load on the FPGA while the CPUs run idle waiting for the results much of the time, or conversely, a design that does not use the FPGA very much at all while the CPUs do most of the work. While this is unfortunate, it is a reality that faces parallel programming every day. Some special solutions for poorly divisible problems are glanced at in Section 6.5.

A reasonable partitioning of the computation between the CPUs and the FPGA has to take into account how the CPUs and the FPGA are going to communicate. There will usually be either two or four threads competing for access to the FPGA, so naturally there has to be some form of arbitration between them. Two different approaches to this problem, with several variants, will be presented next.

6.4.1 FPGA-side Arbitration

With FPGA-side arbitration, most of the responsibility to avoid problems lies on the FPGA. Two somewhat different solutions are presented in Figure 6.2 and Figure 6.3, here displaying a dual-CPU dual-core setup. In the former, the computation element is duplicated in the FPGA, so each core can communicate directly with its own dedicated element. In the latter, an arbitration unit in the FPGA holds off requests from the other cores if the single computation element is already busy with a request from another core.

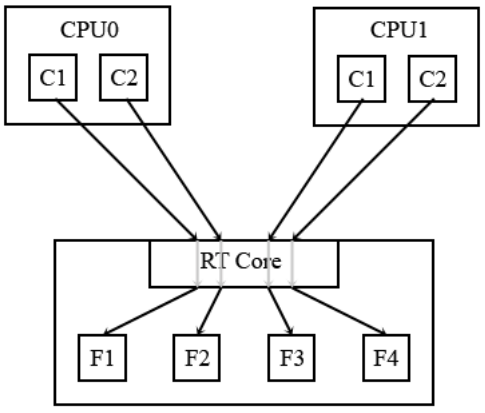

Figure 6.2: Arbitration to duplicated FPGA computation elements.

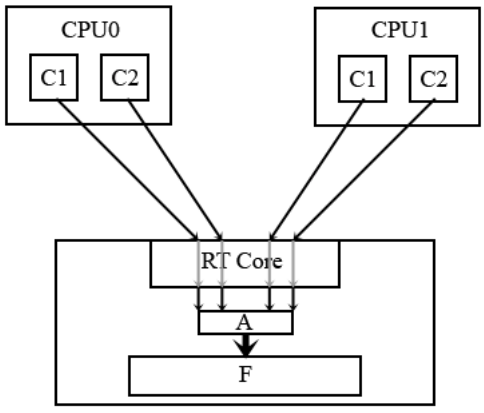

Figure 6.3: Arbitration unit in FPGA handles arbitration to a single computation element.

Both cases depend on the memory area allocated to the FPGA being divided between the cores, so C1 on CPU0 would for instance address 0-32M, C2 on CPU0 would address 32-64M, and so on. That way, the different CPUs would not interfere with each others addressing of the FPGA, and the FPGA can tell which of the CPUs the request originated from. The solution in Figure 6.2 is thus based on the fact that F1 would handle all

requests from C1 on CPU0, on the base that it is written to the 0-32M memory area, and so on. The solution in Figure 6.3 is more complex, and only one core can use the FPGA's computation element at a time, but it allows a much larger element, and will quite possibly lead to a higher utilization of the FPGA at the expense of CPU idle time.

FPGA-initiated writes are also possible using this solution, but would require one FPGA transfer region per core in most cases.

6.4.2 SMP-side Arbitration

SMP-side arbitration, shown in Figure 6.4, is an alternative and simpler solution. In this case, access to the FPGA is arbitrated between the various cores of the CPUs through OpenMP. A single memory map to the FPGA is created, that can be accessed by all the threads, and access is then arbitrated using either the `single`, `master` or `critical` OpenMP directive. The first two directives are used whenever the FPGA operation should only be performed once. If the directive is `single`, it will be performed by the first thread to reach the directive, while if it is `master`, it will be performed by the master thread of the parallel region. `critical` should be used if the FPGA operation is to be performed by all threads, and will enforce access in a way that only one thread accesses the FPGA at a time.

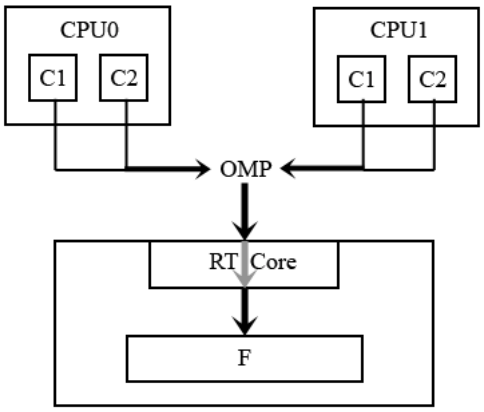

Figure 6.4: Arbitration done through OpenMP.

Compared to the FPGA-side arbitration, SMP-side arbitration places fewer burdens on the FPGA logic with the cost of somewhat more complex code, but this is usually a reasonable compromise, seeing as it is much harder to create FPGA logic than it is to code a parallel program in OpenMP. However, in the cases where the computation element is small enough to be duplicated two (single-core) or four (dual-core) times, the increase in efficiency could be enough to warrant the higher complexity of the hardware development.

A similar concept to this is where the FPGA generates data independent on the CPUs, as is done in the Mersenne Twister Accelerator, a random number generator described in

Section 7.5.3. In this particular case, access arbitration would be local on the SMP, and only control the access of the memory area where the random numbers are stored, not access to the FPGA per se.

6.5 Alternative Solutions

If the problem in question does not easily shoehorn into the Cray XD1 setup, it can in some cases be just as efficient to let parts of the system idle instead of trying to force a parallelization of code that either has very low granularity, or does too little work outside of the FPGAs to warrant the work.

One extreme example was presented in Section 4.3, where only one CPU in the entire chassis was invoked to control the six FPGAs. The communication between the FPGAs themselves was done through the RocketI/O interface, and all the calculations were done mostly independent on the host CPU. If a problem is highly suitable for the FPGAs, and the problem can fit within the constraints of one chassis, this solution is a good compromise.

If a problem is too interdependent to be efficiently split up into relatively independent parts, reducing the implementation to the case of running on a single node is of course possible. Likewise, if the problem is heavily reliant on the FPGA, and one CPU is able to saturate it, reducing the implementation to the case of running single-CPU nodes (where one CPU remains idle) is also an option.

Obviously, if there aren't any tasks that can easily be put on the FPGA, just running the program on the CPUs is always possible, but this somewhat voids the entire point of using a Cray XD1 in the first place.

III The Hardware Developer's Guide to the Cray XD1

This part will introduce the Cray XD1 seen from a hardware developer's point of view. Section 7 introduces the development platform, will mainly focus on the use of the various cores delivered from Cray. Section 8 will take a somewhat more general look on the debugging methods available on this platform. Together they attempt to provide a foundation for developing FPGA designs on the Cray XD1 platform.

7 Creating the FPGA Logic

Creating a design for the FPGAs on the Cray XD1 is a complex task, which in most cases requires solid skills in hardware design using HDL-based tools. Creating FPGA logic can be likened to coding programs in assembly, with a cron job randomly changing a source or target register at a random line every fifth minute. This text does not aim to provide a base foundation for those skills, as the material required could easily fill several books by itself. While Section 7.1 provides a discussion of the development languages involved, with sources for further information aimed at both novice- and intermediate-level developers, most of the remaining text does require at the very least a basic level of knowledge in VHDL and hardware development.

This part will cover the concepts of FPGA development that were glossed over in Part II, illustrated in Figure 7.1 below. While the use of the **`fcu`** tool to prepare the binary file for use with the FPGA could be regarded as a part of the hardware development process, this was covered earlier in Section 5.3 as a part of the software development process, and will not be repeated here.

Section 7.1 will as mentioned discuss various languages that can be used for the FPGA development. Section 7.2 will likewise discuss some of the common tools used for this development. Section 7.3 describes the Cray framework, which defines the IP cores used for communication with the SMP node and the QDR II SRAM. Section 7.4 has a more detailed description of these IP cores, with a high-level description of the interfaces. Finally, Section 7.5 introduces the three example programs that are supplied with the Cray XD1.

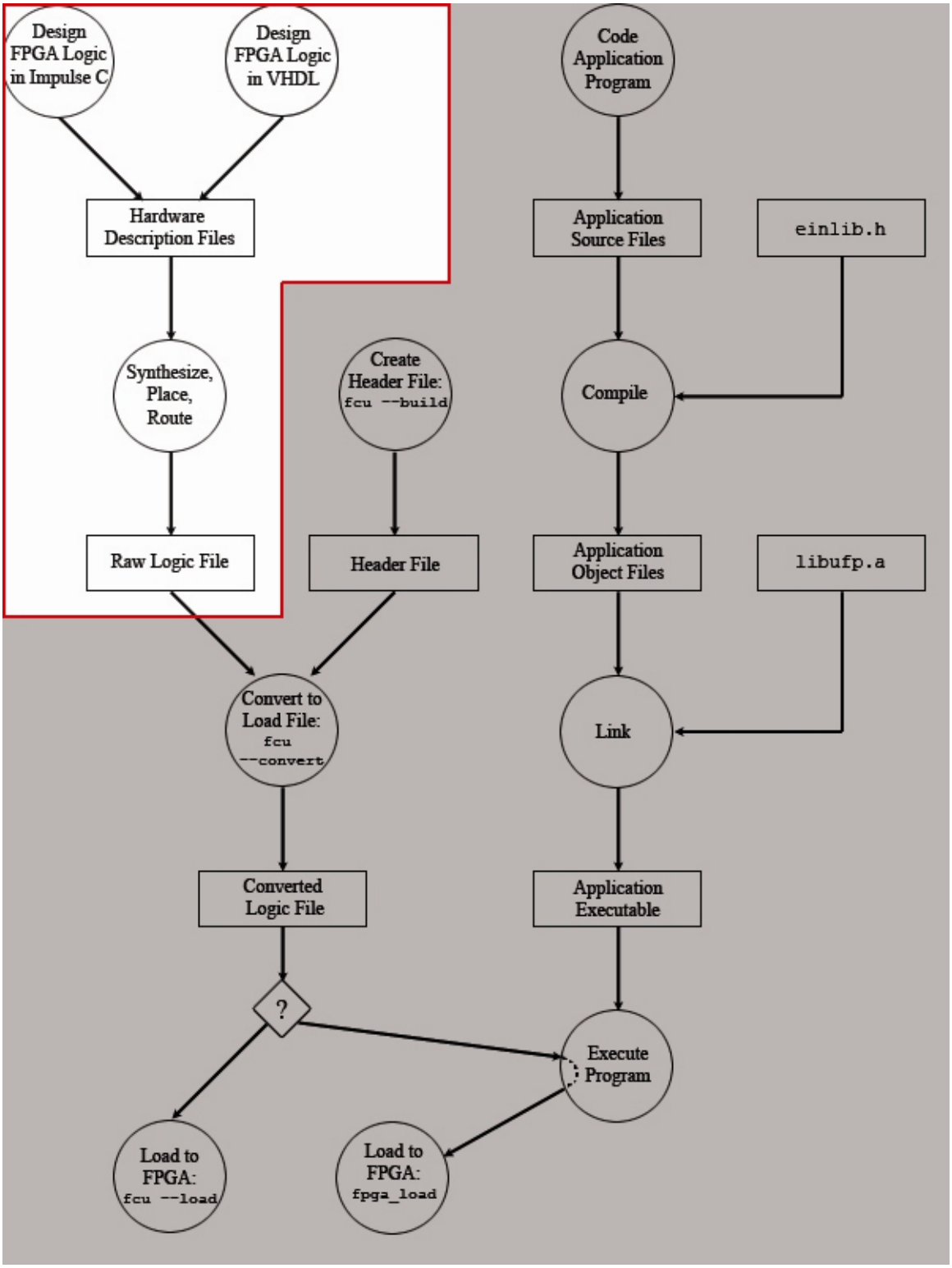

Figure 7.1: Typical FPGA hardware development workflow.

7.1 Regarding VHDL and C-based Development

VHDL[19] is one of the most popular languages available for describing the behavior of hardware. It was originally developed by the US Department of Defense as a way to document the behavior of ASICs used in equipment from suppliers. *Logic simulators* that could simulate the behavior described by VHDL were soon developed, followed by *logic synthesis* tools that could create a physical representation of the circuit described. The first version of the standard, designated IEEE standard 1076-1987, was further refined in 1993 [VHDL87, VHDL93]. An extension, designated IEEE standard 1164, extended VHDL with a uniform representation of non-standard logic values [WIKI07].

While writing VHDL directly is still the most common approach to developing hardware on FPGAs, there are alternative approaches to the task. C-based tools have been developed, both because VHDL can become quite verbose, and because the learning curve is relatively sharp. Two examples are SystemC[20] and the more recent Impulse C[21]. These tools will typically compile the C/C++ code into VHDL or Verilog[22], which can then be processed by the vendor-specific FPGA tools.

There are many books written about VHDL, such as [YALA98], which is a beginner's guide, and [ASH02], which is more comprehensive and can be used as a reference. Many online guides to VHDL also exist, but the quality varies. Much information about SystemC is available at their website, while the authoritative source of information on Impulse C is [PELLE05]. C/C++-based tools will however not be discussed further here, due to lack of compiler availability and support.

7.2 Development Tools

FPGAs are implemented in significantly different ways, so the particulars of the synthesizing, component placement and signal routing vary both between vendors and the particular products each vendor provides. To be able to convert a set of VHDL files to something that can be uploaded to a FPGA, vendor-specific tools are therefore required. *Xilinx ISE Foundation*[23] is the most common tool for development on the Xilinx FPGAs, and is available for 32- and 64-bit Linux, Solaris and Windows. A free version called *ISE WebPACK* is available from Xilinx' website [XIDL], but it does not support the Virtex-II Pro XC2VP50-7 fitted on the Cray XD1 as of this writing, so the full version has to be obtained. A cheaper and less feature-rich product called *ISE BaseX* that supports this particular FPGA is also available.

[19] **V**HSIC **H**ardware **D**escription **L**anguage. VHSIC is an acronym for Very High Speed Integrated Circuit.
[20] http://www.systemc.org/
[21] http://www.impulsec.com/
[22] Verilog is another popular HDL that is somewhat reminiscent of C.
[23] The particular version used while writing this text was *Xilinx ISE Foundation 7.1i* with Service Pack 4.

WARNING: Make sure to get the newest version of ISE Foundation. The unpatched version of ISE Foundation 7.1i has been confirmed to fail "randomly" when synthesizing designs that use the Cray IP cores, creating a variety of problems, in particular with timing. During testing, these problems were also indicated by consistent RT Rx Bus Parity Errors, reported by the RT core.

Several other tools used to support the development of FPGA designs are also provided by Xilinx. Most notable of these are *ModelSim*, a simulation tool used for debugging, and *ChipScope Pro*, used to perform live debugging through the FPGA's *JTAG* interface. These two tools are discussed in Sections 8.1 and 8.2, respectively.

7.3 Using the Cray Framework

The default installation comes complete with a framework that is ready to be deployed against the Cray XD1 FPGA. Among other things, it includes the RapidArray Transport Core and the QDR II SRAM Core, described further in Section 7.4. It has a complete definition of all external FPGA interfaces and pins, and is complete with the needed makefiles for building and simulating the design. The structure of the framework is shown in Figure 7.2.

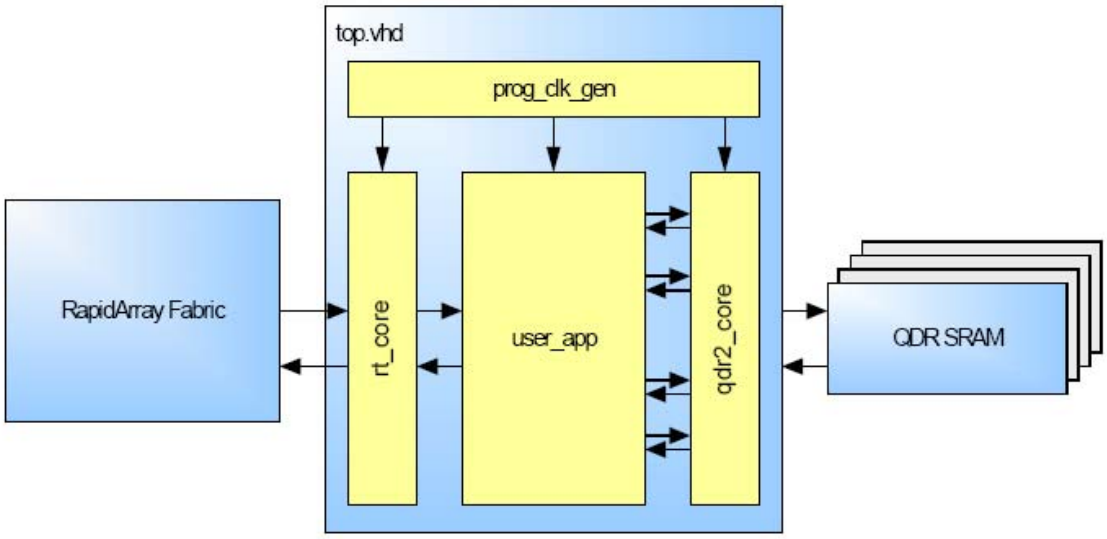

Figure 7.2: Cray XD1 Framework [S-6400].

`user_app` is here the user-created application, which is linked in through the framework, and will usually be the only part modified by the user. `rt_core` and `qdr2_core` are the interfaces to the RapidArray Fabric and the QDR II SRAM, respectively. `prog_clk_gen` generates the various clock signals needed by the two cores, and also provide a clock signal to `user_app`.

7.3.1 Directory Structure

The framework is located under `/opt/ufpapps/vhdl_template/src`, but is also dependent on the Cray cores located under `/opt/ufpapps/libxc2vp`. To do any work with them, the best solution is simply making a copy of the entire `/opt/ufpapps` directory to your home. Doing so will also copy over the example designs provided by Cray, which are further described in Section 7.5.

The directory structure of the framework itself is indicated in Table 7.1 below.

Directory	Subdirectory	Description
`(root)`	`/`	Contains the "master" makefile, which invokes various makefiles in the subdirectories to perform tasks, as well as a file containing variables included by the other makefiles. Also holds the framework changelog.
`hdl`	`/`	Contains the `top.vhd` file that fully defines the interfaces between the Cray cores and the user application, as well as a makefile used to compile the VHDL files for simulation.
	`/cray`	Contains the VHDL code for the Cray cores.
	`/user_app`	Contains the VHDL code for the user application.
`hdl_tb`	`/`	Contains a VHDL test bench for the entire design, as well as a behavioral model for the RT Core and source model for the QDR II SRAMs, as well as a makefile to simulate the test bench.
`par`	`/`	(Empty directory)
	`/xc2vp50`	Contains the various synthesis, implementation and constraint files for the FPGA module, as well as a makefile to build the design. Used mainly by IIS Foundation.
`sim`	`/`	Contains ModelSim scripts that can be used to simulate the design.
	`/tc_XX`	A numbered series of directories that contain test case input and output files.
`simlib`	`/`	(Empty directory)
	`/modelsim`	Contains simulation libraries for Modelsim.
	`/riviera`	Contains simulation libraries for Riviera.

Table 7.1: Directory structure under `/opt/ufpapps/vhdl_template/src/`*`PARTNUM`*`_vhdl`

7.3.2 Working with the Framework

There are two basic ways of operating with the framework: through the GUI of IIS Foundation, and through the CLI using standard text editors such as *Emacs* or *Vim*. The

choice depends on the user's familiarity with the platform and personal preference, but in general the GUI is simpler, while the CLI can be more powerful for an experienced user.

7.3.2.1 Using the GUI

To modify an instance of the framework through ISE Foundation, simply open the `top.npl` file present in the `par/xc2vp50` directory. Depending on your version of ISE Foundation, the project will likely have to be converted to correspond with the new version, a process that ends up turning the `top.npl` file into `top.ise`. This process is automatic, and should not cause any problems.

After the project has been opened, you should see a view much like the one in Figure 7.3. Note that this is taken from the Windows version, and some details could vary between this and the Linux version.

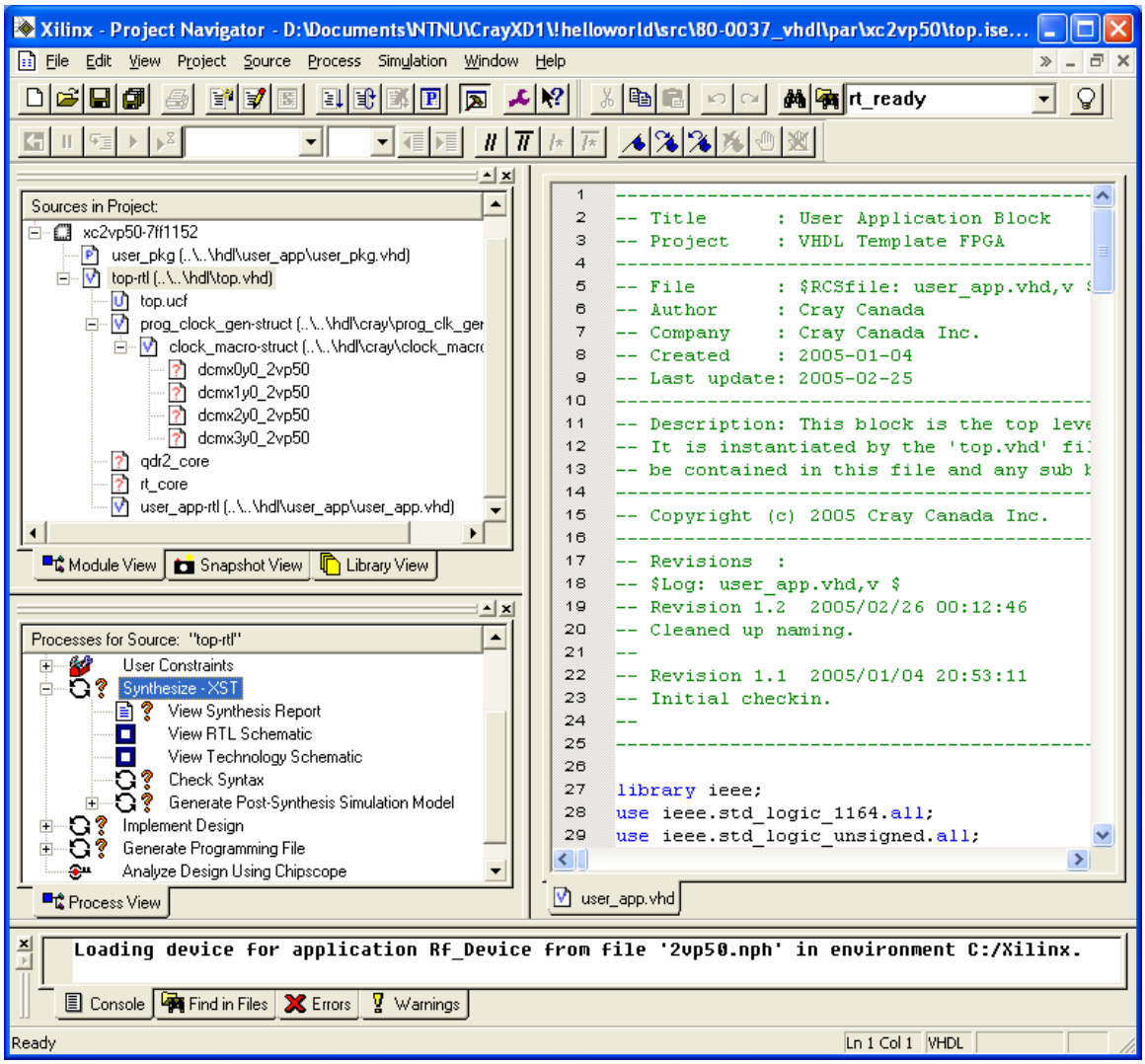

Figure 7.3: The ISE Foundation GUI.

The Module View on the left represents the various modules present in the project. The Process View located directly below it is connected to the Module View, and the choices available here vary depending on which module is selected[24]. For instance, if a VHDL file is selected, you will be given choices like *Synthesize*, which includes functions like *Check Syntax*, shown on the figure. The editor window on the left is where files are edited. On the figure, the `user_app.vhd` file is currently being modified. The bottom-most window is the Console, where various output from the tools invoked by the GUI is displayed. The output can also be filtered by Errors and Warnings using the corresponding buttons.

The particulars of creating VHDL files, organizing them into modules and linking them together is outside the scope of this text, but is presented in virtually all texts on VHDL, and some examples are also available in the example designs presented in Section 7.5. As soon as a design has been created, it can be processed into a binary file through the *Generate Programming File* command. Be careful that the topmost VHDL file is selected in the Module View when doing this however, or the process will fail, usually during the *Map* stage. Provided that the operation is successful, the finished binary file will be put in the `par/xc2vp50` directory as `top.bin`.

This short introduction only grazes on a few of the core functions of the GUI. The full manual presenting its use is available at Xilinx' website [XILIB].

7.3.2.2 Using the CLI

Since many people prefer to work in a strictly CLI-based environment, this is also made possible by ISE Foundation. The editor provided with the GUI version of ISE Foundation is horrible at best, so editing files directly in editors such as *Emacs* or *Vim* can be very efficient for people who are familiar with their use. Modules are also available to extend Emacs with VHDL capability, such as [EMVHDL].

The basic workflow of CLI development is to work on the files in the `hdl/user_app` directory, by creating the needed modules and linking them together through the `user_app.vhd` and `user_pkg.vhd` files. All the source files then has to be included in the `SOURCE` variable in the makefile present in this directory.

All the tools required for building and simulating designs are available from the CLI, but the arguments needed are relatively complex. Cray has therefore provided a number of makefiles that can be modified to fit particular tasks; these are included in the various directories as described in Section 7.3.1. Several targets can be given to the makefiles. The most important of these is `xc2vp50`, which will produce the FPGA binaries by running the various ISE Foundation tools in the correct order. Other targets include `sim_setup` to setup the various libraries and directories used for simulation, and `sim`

[24] To select a module, single-click it. To open it in the editor, double-click it.

and `sim test=dir` for running batch mode and interactive mode simulation, respectively. For more information about the various make targets and files used during CLI development, see [S-6400 pp. 42-44].

Note that the CLI can also be used from Windows, for example by using *Cygwin*[25], a Linux-like environment for Windows that acts as a Linux API emulator, allowing you to run many of the most common GNU tools.

7.4 Cray IP Cores

The Cray XD1 is delivered with two major IP cores, namely the RapidArray Transport Core used to interface with the SMP node, and the QDR II SRAM Core used to interface with the local SRAM. Both of these cores are hooked up with the framework discussed earlier in Section 7.3. This section will discuss how to use these particular cores to interface with their respective systems, including the various signals and timing information required. Complete implementations are too verbose to include here, and could not be fully tested due to the problems described in Appendix C, but examples of such implementations are included with the example programs, such as the `rt_client.vhd` and `qdr2_if.vhd` files included with the *Hello World* program, further discussed in Section 7.5.1.

All signals are active-high (active on logic '1') unless otherwise stated. Important exceptions are the `user_reset_n` signal from the RT Core as well as the read and write strobes on the QDR II SRAM Core, all of which are active-low (active on logic '0'). Note that most active-low signals can be recognized by them being suffixed with a '_n'.

7.4.1 The RapidArray Transport Core

The RapidArray Transport Core is used by the user FPGA logic to interface with the SMP node. It offers a 199MHz, 64-bit wide bus, allowing a sustained speed of up to 1.422GBps simultaneous transfer to and from the fabric when overhead is included. Buffers that can hold up to 32 requests or responses allow the core to bridge between the user logic clock domain and the RapidArray bus clock domain, and burst writes of up to nine quadwords are supported.

The interface is split into two major parts: the Fabric Request Interface, which is used when dealing with CPU-initiated transfers, and the User Request Interface, which is used for FPGA-initiated transfers. The interface is shown in Figure 7.4 below. The three other signals shown in the figure are `user_clk` and `user_enable`, which are provided to the core by the framework, as well as `user_reset_n` which is an active-low reset signal triggered by `fcu -r` as well as `fpga_reset` in the C API. Not shown on the

[25] http://www.cygwin.com

figure is the signal `rt_ready`, which indicates to the user logic that the RT Core is ready. Several other signals are also present in the VHDL file describing the function of the RT Core, but their use (if any) is not documented, and is therefore unknown.

The information given here about the RT Core is a condensed version of the information given in [S-6411], which also includes a number of timing diagrams as well as additional information omitted here.

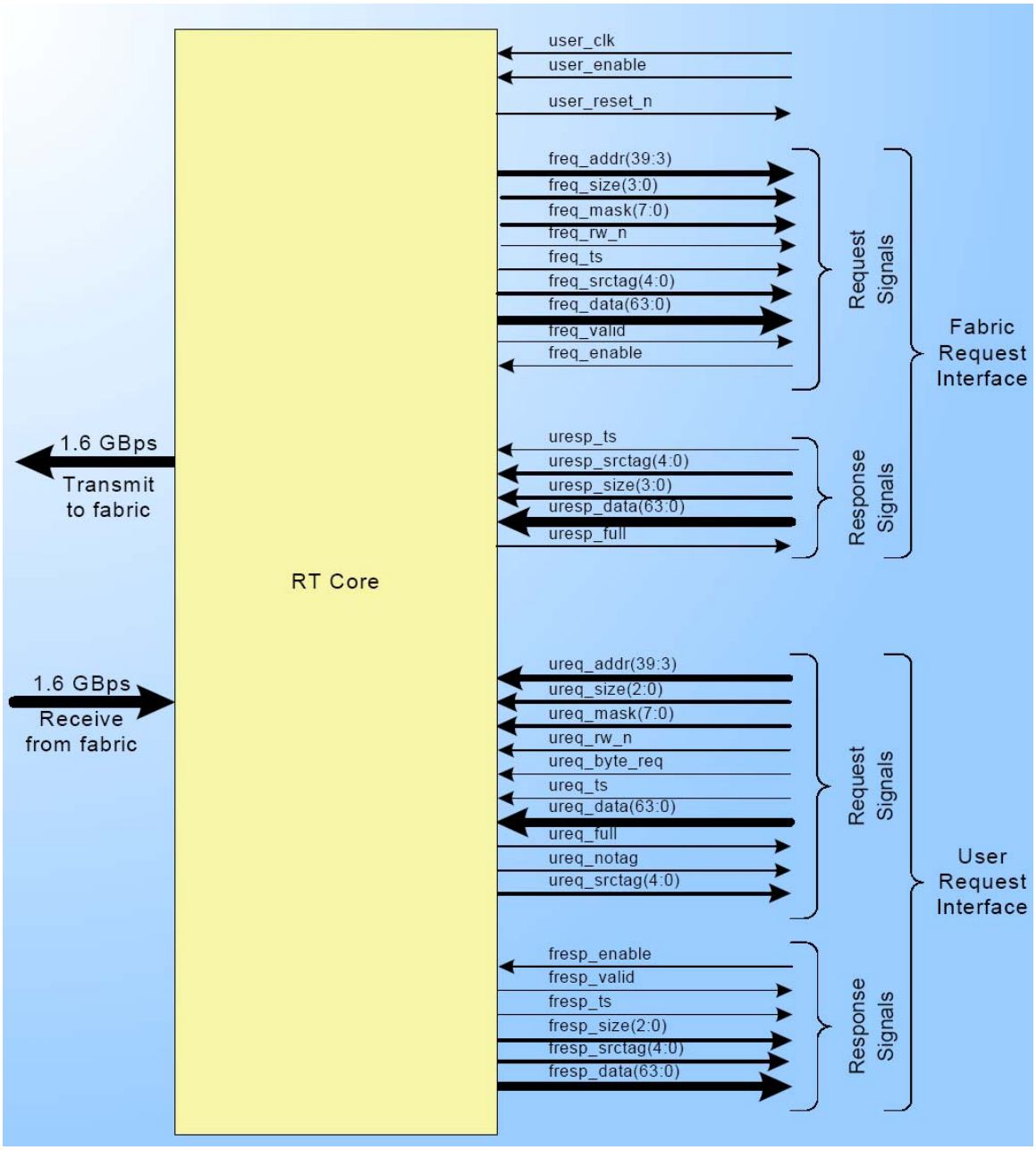

Figure 7.4: The RT Core Interface [S-6411].

7.4.1.1 The Fabric Request Interface

The Fabric Request Interface provides a way for the SMP node to request and write data to and from the FPGA. It consists of a set of request signals (`freq`) and response signals (`uresp`), where the former forwards the details of the request to the user design, with elements such as address, size, byte mask, and whether it is a read or write request, while the latter is used by the user design to respond to the request.

The signals used are as follows:

Fabric Request Signals

Name	*Driver*	*Description*
`freq_addr(39:3)`	RT	Address for request. The 3 lowermost bits are ignored due to 64-bit quadword addressing.
`freq_size(3:0)`	RT	The size of the request in 64-bit quadwords.
`freq_mask(7:0)`	RT	Byte mask for data conditioning.
`freq_rw_n`	RT	Set to logic '1' for reads or logic '0' for writes.
`freq_ts`	RT	Indicates the first cycle of a new request
`freq_srctag(4:0)`	RT	Identifier for read requests.
`freq_data(63:0)`	RT	Data to be written during write cycles.
`freq_valid`	RT	Indicates if signals from the interface are valid.
`freq_enable`	User	Allows the RT Core to drive a new request from the buffers next cycle, if available.

User Response Signals

Name	*Driver*	*Description*
`uresp_ts`	User	Notifies the RT Core of the first cycle of a new response.
`uresp_srctag(4:0)`	User	Identifier for read requests, echoed from the request.
`uresp_size(3:0)`	User	The size of the response in 64-bit quadwords.
`uresp_data(63:0)`	User	The response data.
`uresp_full`	RT	Indicates that the response buffers are at least 75% full, and that the user logic should hold off new responses until the signal is de-asserted.

When a new request is posted to the user logic, the signals `freq_valid` and `freq_ts` will be enabled. `freq_rw_n` will indicate whether the request is a read or a write. The request will be present on the interface as long as `freq_enable` is de-asserted, but will be replaced with the next request on the cycle following this signal's assertion provided there are any further requests in the buffer.

In case of a write, `freq_data(63:0)` contains the data to be written, conditioned by `freq_mask(7:0)` which is used to indicate valid bytes used for the write (a logic '1' indicates a valid byte). `freq_addr(39:3)` contains the address for the first quadword of the write. `freq_size(3:0)` is used to indicate the number of quadwords written in case of a burst, where valid values are from 0x0 for a single quadword to 0x8 for nine

quadwords. freq_srctag(4:0) is not valid for write requests, as these should not be responded to. If the request is a burst write, the data content will change to give a new quadword every clock cycle.

In case of a read, freq_addr(39:3) contains the address for the first quadword of the read, while freq_size(3:0) indicates the number of quadwords requested. freq_srctag(4:0) provides the user logic with an identification tag, used later to identify the request when responding. freq_data(63:0) and freq_mask(7:0) are not valid during a read request.

Only read requests require a response. To make a response to the fabric, uresp_srctag(4:0) must be driven with the tag obtained from during the request. uresp_size(3:0) must be driven with the size of the response in quadwords, where valid values are from 0x0 for a single quadword to 0x8 for nine quadwords. uresp_data(63:0) must be driven with the data content of the response. For burst writes, the data content must be driven with a new quadword every cycle. After a write or series of writes, the data content must be driven with the last value written for one additional cycle.

7.4.1.2 The User Request Interface

The User Request Interface provides a way for the FPGA to request and write data to and from the SMP node. It consists of a set of request signals (ureq) and response signals (fresp), where the former is used by the user logic to make requests to the fabric, with elements such as address, size, byte mask, and whether it is a read or write request, while the latter is used by the fabric to respond to the request.

The signals used are as follows:

User Request Signals

Name	*Driver*	*Description*
ureq_addr(39:3)	User	Address for user request. The 3 lowermost bits are ignored due to 64-bit quadword addressing.
ureq_size(2:0)	User	The size of the request in 64-bit quadwords.
ureq_mask(7:0)	User	Byte mask for data conditioning.
ureq_rw_n	User	Set to logic ‘1’ for reads or logic ‘0’ for writes.
ureq_byte_req	User	Asserted if the ureq_mask(7:0) bus should be used.
ureq_ts	User	Asserted to indicate the first cycle of a new request
ureq_data(63:0)	User	Data to be written during write cycles.
ureq_srctag(4:0)	RT	Identifier for read requests, driven by the RT Core to allow the user logic to identify fabric responses.

ureq_full	RT	Indicates that the request buffers are at least 75% full, and that the user logic should hold off new requests until the signal is de-asserted.
ureq_notag	RT	Indicates that there are more than 24 outstanding source tags, and that the user logic should hold off new read requests until the signal is de-asserted.

Fabric Response Signals

Name	*Driver*	*Description*
fresp_enable	User	Allows the RT Core to drive a new response from the buffers next cycle, if available.
fresp_ts	RT	Indicates the first cycle of a new response.
fresp_srctag(4:0)	RT	The tag used by user logic to identify the response.
fresp_size(2:0)	RT	The size of the response in 64-bit quadwords.
fresp_data(63:0)	RT	The response data.
fresp_valid	RT	Indicates if signals from the interface are valid.

In the case a write, ureq_full should be examined, and the write should be held off if this signal is asserted. ureq_rw_n must be driven to 1, and ureq_addr(39:3) must be driven with the target address for the first quadword of the write. ureq_size(2:0) must be driven with the number of quadwords written, where 0x0 indicates a single quadword while 0x7 indicates eight quadwords. Note that this bus is three bits wide on the User Request Interface, while the corresponding bus on the Fabric Request Interface is four bits wide.

The data to be written must be driven to ureq_data(63:0). If byte masking is required, ureq_byte_req must be driven to 1, while ureq_mask(7:0) must be driven with the required mask. If ureq_byte_req is driven to 0, ureq_mask(7:0) will be ignored by the RT Core. ureq_ts must then be asserted to indicate the first cycle of the new request. For burst writes, a new quadword must be driven to ureq_data(63:0) every cycle.

In the case a read, ureq_notag should be examined, and the read should be held off if this signal is asserted. ureq_rw_n must be driven to 0, and ureq_addr(39:3) must be driven with the source address for the first quadword of the read. ureq_size(2:0) must be driven with the number of quadwords read, where 0x0 indicates a single quadword while 0x7 indicates eight quadwords. The RT Core provides the tag used to identify the response on ureq_srctag(4:0), so this value must be saved for later use. ureq_ts must then be asserted to indicate the request. The other signals are not used during a read.

When a response is ready, and fresp_enable is asserted by the user logic, the RT Core will drive fresp_size(2:0) with the size of the response, and fresp_data(63:0) with the first quadword of the response. fresp_valid should be evaluated to make sure the signals are valid, and the fresp_ts strobe will be asserted for one cycle to indicate the start of the response. fresp_srctag(4:0) will

be driven with the same value as was given the user logic during the request. If the response is a burst, then `fresp_data(63:0)` will be driven to a new quadword every cycle.

7.4.1.3 Using the Fabric and User Request Interfaces Efficiently

To be able to attain a performance close to the maximum for the RT interface, something which is especially important for I/O-bound problems, extensive use of the burst facilities provided by the RT Core is required, as any sequence of writes from the FPGA to the SMP node requires a one cycle cooldown regardless of the number of writes performed. This is true both for performing writes through User Request and through Fabric Response [S-6411 page 11, 14].

Also, particular notice should be paid to the fact that while the User Request Interface has the ability to both burst reads and writes, the Fabric Request Interface can only burst writes. A "write-only" architecture is therefore often recommended, where the SMP node writes the data to the FPGA, and the FPGA writes the results back to the SMP node.

7.4.2 The QDR II SRAM Core

The QDR II SRAM Core is used by the user's FPGA logic to interface with the external SRAM memory on the FPGA expansion board. There are four separate and independent dual-ported SRAM chips, each capable of transferring 1.6GBps in each direction, for a total of 6.4GBps. The busses between the FPGA and the QDR II SRAM are actually 36-bit wide double data rate busses, but the QDR II SRAM Core represents these as a single data rate 72-bit wide bus.

The SRAM Core reflects this configuration in that it has four independent interfaces, one for each of these chips, as indicated in Figure 7.5 below. Four of the five remaining signals shown, namely `reset_n`, `qdr_clk0`, `qdr_clk90` and `locked_dcm` are provided by the framework, and will usually not be of any concern to the user. The remaining signal, `ram_rdy` (renamed `qdr_ready` for the `user_app` part of the framework) indicates to the user logic that the QDR II SRAM Core is ready. Like the RT Core, several other signals are also present in the VHDL file describing the function of the QDR II SRAM Core, but their use (if any) is not documented, and is therefore unknown.

The information given here about the QDR II SRAM Core is a condensed version of the information given in [S-6412], which also includes a number of timing diagrams as well as additional information omitted here.

Important note: When the QDR II SRAM Core is used, the FPGA must operate at a minimum clock speed of 130MHz. If lower speeds are required, then user logic must be used to bridge the clock domains.

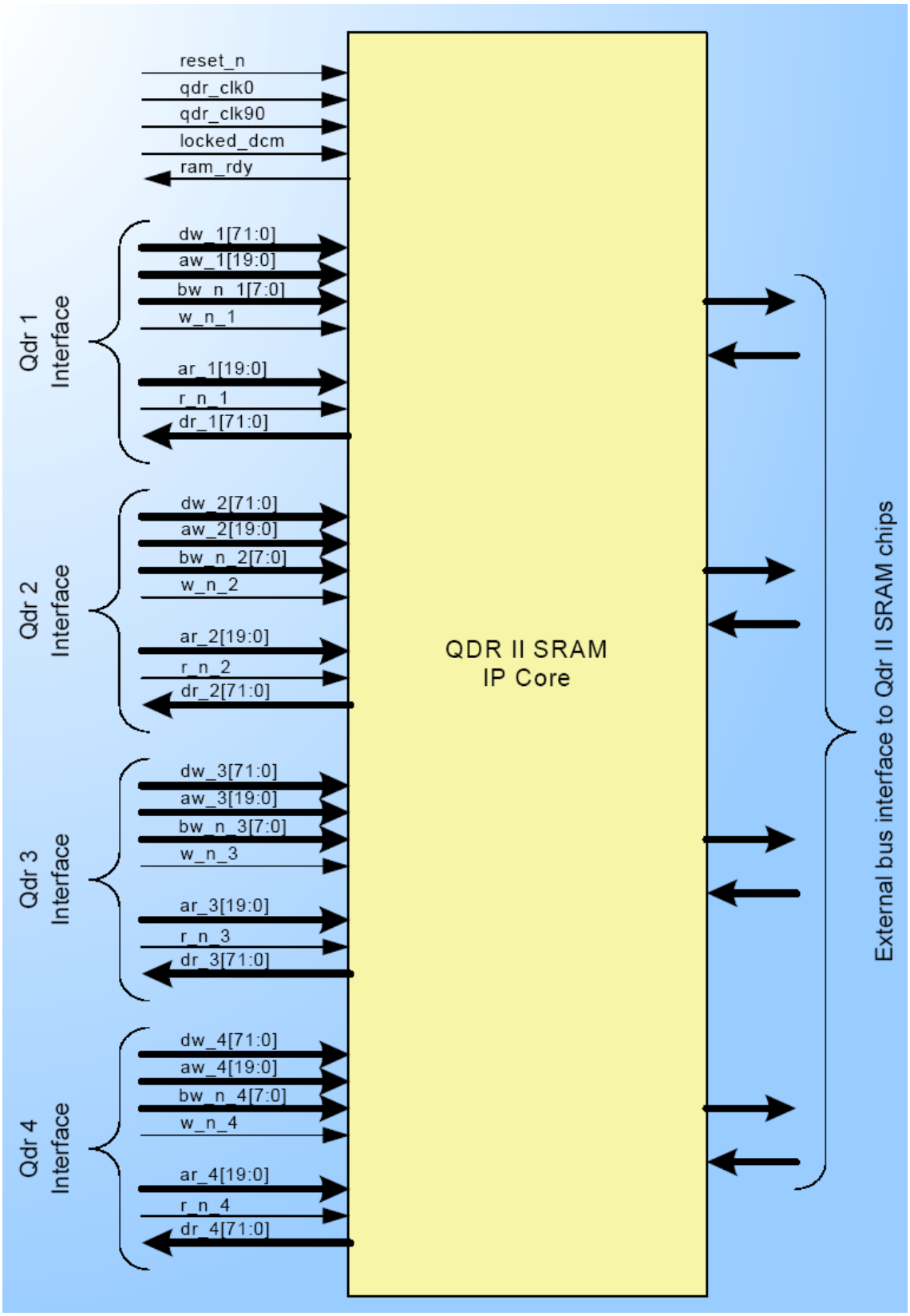

Figure 7.5: The QDR II SRAM Core Interface [S-6412].

7.4.2.1 The QDR II SRAM Interface

The four interfaces for each of the physical SRAMs are identical, so the information will not be repeated four times. In the signal description below, capital X should be replaced with the number (1-4) of the SRAM that is to be accessed. Note that since the SRAM is dual-ported, there are separate address and data lines for read and write, as well as separate read and write strobes.

Name	*Driver*	*Description*
`r_n_X`	User	Active-low read strobe.
`ar_X(19:0)`	User	Read address.
`dr_X(71:0)`	QDR	Read data.
`w_n_X`	User	Active-low write strobe.
`aw_X(19:0)`	User	Write address.
`dw_X(71:0)`	User	Write data.
`bw_n_X(7:0)`	User	Active-low bytewise write data mask.

7.4.2.2 Writing data to the QDR II SRAM

The user logic can write 72 bits, usually 64 bits data and 8 bits parity, to each QDR II SRAM every cycle, with no restrictions on bursting or address. `aw_X(19:0)` must be driven with the target address for the write, while `dw_X(71:0)` must be driven with the data to be written. `bw_n_X(7:0)` is an active-low mask used to determine which of the bytes in `dw_X(71:0)` should be actually written. Note that each bit of `bw_n_X(7:0)` actually enables or disables 9 bits due to the parity support, so `bw_n_X(0)` enables `dw_X(8:0)`, and so forth.

In addition to setting the address, data and mask busses, `w_n_X` needs to be asserted. Note that the signal is active-low, so setting it to logic ‘0’ will trigger the write.

7.4.2.3 Reading data from the QDR II SRAM

The user logic can also read 72 bits, usually 64 bits data and 8 bits parity, from each QDR II SRAM every cycle, with no restrictions on bursting or address. `ar_X(19:0)` must be driven with the source address for the read, and the `r_n_X` strobe needs to be asserted. The strobe is active-low like its cousin, so setting it to ‘0’ will trigger the read.

The result will then be driven to `dr_X(71:0)`. However, there is currently a read latency of eight clock cycles from when the read is issued to the data is available and stable on the `dr_X(71:0)` data line, so the user logic has to take this into account.

Note that if a read and a write are issued to the same address at the same time, the read will return the data that was written by the write, not the previous data from that address.

7.5 Example Programs

Cray provides three example programs, complete with the source files for both the software and the hardware parts of the implementation. The simplest of these programs is Hello World, which writes some data to the FPGA and then reads it back. Mince is somewhat more complicated, and is used to test the memory and bus interfaces more thoroughly than Hello World. Finally, Mersenne Twister Accelerator is an implementation of the Mersenne Twister random number generator.

7.5.1 Hello World

The Hello World example program, located at `/opt/ufpapps/hello`, provides an example of simple communication between the SMP node and the FPGA. The basic functionality provided is writing to and reading from a set of user-defined registers in the FPGA, writing to and reading from an internal BlockRAM and writing to and reading from one of the FPGA's QDR II SRAM devices.

Detailed information about the implementation can be found at [PNR-DD-0023].

7.5.2 Mince

Mince, or Minimal Compute Engine, provides a system intended for performing sanity checks on the FPGA and FPGA-related interconnects and devices. Its base functions include Bit Error Rate Test (BERT) blocks to exercise the QDR II SRAM interface and RapidArray Transport interface, a bridging function that allows an SMP access to the FPGA's local QDR II SRAM, and a set of registers to configure and monitor the device.

Two programs are included with the program. `berttest` invokes the RT and QDR BERT blocks for a specified amount of time. `qdrtest` invokes the QDR bridging function, and runs various tests on the QDR II RAM.

Detailed information about the implementation can be found at [PNR-DD-0015].

7.5.3 Mersenne Twister Accelerator

The Mersenne Twister Accelerator (MTA) is an implementation of the popular Mersenne Twister random number generation algorithm. The implementation generates the random numbers on the FPGA, and then writes them directly to a buffer in the SMP node's DRAM, making it as fast as reading a value from local DRAM for the SMP node to obtain random numbers at will.

Detailed information about the implementation can be found at [PNR-DD-0022].

8 Debugging the FPGA Logic

Debugging FPGA logic is not an easy task. Usually, simulations on several levels will be performed, such as pure VHDL behavioral simulation, as well as simulation on the logic after it has gone through the various implementation phases. Even so, timing errors could occur when the logic has been loaded to the FPGA. Therefore, several different debugging strategies are required.

Section 8.1 discusses using the simulation tool ModelSim from Mentor Graphics to simulate the behavior of the FPGA logic before it is actually put on the FPGA. Section 8.2 discusses the logic analyzer ChipScope Pro, which can be used to analyze the logic after it has been uploaded to the FPGA, through the FPGA's JTAG debug port. Since many software developers often like debugging through simple *printf* statements, and since it is not always possible to run the logic through a debugger, Section 8.3 takes a quick look at how something similar can be done on the FPGA.

The Cray example designs also come complete with models for a simulator called Riviera from Aldec Inc.[26], of which versions exist for Linux, UNIX and Windows, but this tool will not be discussed here.

8.1 Debugging with ModelSim

ModelSim is as earlier mentioned a simulator tool that can be used to simulate the FPGA logic at various stages of design. A free version is available from Xilinx' website [XIDL], but it is much slower than the licensed version for large designs, so a full copy should be obtained for large-scale logic designing. The cores provided with the Cray XD1 are delivered complete with ModelSim models, and a behavioral model for the RT fabric is also available.

ModelSim can be invoked directly from the ISE Foundation GUI, by first creating a test bench/waveform and entering the input values for the simulation. The simulation can then be run on the VHDL behavioral level or after each of the Translate, Map, and Place & Route phases. Doing so will provide the expected output values for that particular input. ModelSim can also be invoked directly from the CLI using VHDL test bench files. There approaches are in this context typically used to simulate separate components or a set of components, as a form of unit testing.

More particular for the Cray XD1 is its use together with the RT behavioral model, as described in [S-6400 pp. 54-56]. Basically, a file of textual input is given that states various read and write commands to the RT fabric, which are then translated by the RT Core and forwarded to the user logic. This, together with the simulation model for the QDR II SRAM Core, enables a user to simulate a complete working system.

[26] http://www.aldec.com/

Concrete examples of this use can be found in the documentation for the example programs, described in Section 7.5.

8.2 Debugging with ChipScope Pro

ChipScope Pro is as earlier mentioned a logic analyzer that can be used to observe the inner workings of an FPGA as it runs. Unlike ModelSim, which is an offline simulation tool, ChipScope Pro needs the computer to be hooked up to the FPGA directly through a so-called JTAG Boundary Scan port. This is a serial port that enables the user to control and observe the FPGA in a variety of ways through cores generated by the ChipScope Pro Core Generator and inserted into the HDL, or directly injected into the netlist by the ChipScope Pro Core Inserter Tool. The ChipScope Pro Analyzer software can then monitor and alter the internals of the FPGA through these cores.

Due to a series of hardware and connectivity problems with the computer hooked up to the JTAG port on the Cray XD1 installation at NTNU, combined with the problems described in Appendix C, the tests that were to be performed with debugging through ChipScope Pro had to be postponed, and were not ready to be included in this text. The reader is therefore referred to ChipScope Pro's manual, available at Xilinx' website [XILIB].

The standard for the JTAG circuitry, IEEE1149.1-1990, is described in [IEEE1149].

8.3 Debugging from the C application

To better facilitate debugging, instrumenting the FPGA logic by adding a variety of debug and status registers that are accessible from the C application is often wise. The amount of logic required for these registers is usually negligible, while the information gained by reading these registers after a failure is detected in the middle of a major computation can be invaluable, especially if it is a recurrent but not readily reproducible error that isn't easily caught on a debugger.

The recommended practice is to create a separate debug module that intercepts memory reads directed to specific memory addresses, and returns the debug information assigned to that address in the desired format. Obviously, the amount of information that can be stored and retrieved from the module depends on its complexity. The simplest case is a design where all signals targeted for snooping are forwarded to the module, and the most recent signals are stored and can be read through the appropriate API calls. More complex designs can store the signals from a number of cycles, or could even write a log to the SRAM or the SMP node's DRAM directly, with the additional cost of logic and communication this entails.

If the debug module is based on saving one or a limited number of signals, as will usually be the case, there should also be some form of error detection employed that stops the

debug module from overwriting the data at the time the error occurred with the data generated in the cycles after the error. Of course, it is not always easy to pin down exactly what should trigger such a stop in the data capture, unless a particular bug is being hunted down and the effects of the bug are well understood, so in many cases the only information available will be regarding the FPGA's state after the error, instead of during, which would be preferred. This could be solved with the logging approach, but said limitations on communication applies, as a maximum of eight bytes can be written to the RT fabric and each of the SRAM devices every cycle, and much of this is likely already consumed by the rest of the FPGA logic.

An example debug module was to be included in this section, but due to the problems described in Appendix C, and the time constrains these imposed, this had to be dropped.

(This page was intentionally left blank.)

Conclusions and further work

An introduction to FPGAs in general was given in Part I. This was followed with an introduction to the use of the Cray XD1, both for software developers in Part II and hardware developers in Part III. Different aspects of the Cray XD1 development cycle were explored, including the C API, a higher-level overview of using the system in a mixed-mode MPI and OpenMP setting, and the particulars of hardware design and debugging.

Many of these topics could be further examined in much greater detail. Some suggestions for further study are listed below, but this list is far from exhaustive.

Section 5.4.3 discusses several issues with the stability and security of the Cray XD1 platform. Further studies could be performed with the goal of charting the severity of these problems, and developing countermeasures against them. One possible avenue could be to create an extension to the RT Core that provides protection against these problems, by disallowing writes outside of memory areas created with `fpga_set_ftrmem`. Wrappers to the API could be created to help enforce these rules, and safeguards could also be added here to avoid FPGA timeouts in the case of FPGA accesses while the FPGA is in the reset state.

Section 6 describes the approach of using mixed-mode MPI and OpenMP to perform computations in parallel over several CPUs and nodes. Studies similar to the one in [SMITH01] could explore how well this approach works compared to, for instance, a pure MPI implementation. Similarly, SMP-side arbitration using MPI alone could be explored.

Section 6.4 explores three different arbitration variants for controlling shared access to the FPGA between several threads. Further work could be done here to create example implementations and evaluate the attainable performance using the different variants, both with regard to the maximum size of the computation elements (which would be different on the single element and the duplicated element implementation), the utilization of the elements, and the speedup attained.

Section 8.2 gives an introduction to debugging techniques on the Cray XD1 using ChipScope Pro on the JTAG debugging interface, but due to a series of hardware failures on the computer that was set up for this debugging, this was not accomplished to the degree intended, and could therefore be performed after the necessary hardware is in place.

Further readings

The official Cray documentation is a good starting point for further information about the Cray XD1 system. In particular, [S-2433] contains much information about software development, while [S-6400] is intended for the developers of the FPGA logic.

Software designers that seek to develop FPGA logic should seek out books on the subject, such as [MAX04] for information on the FPGA itself and [YALA98] for an introduction to VHDL.

Glossary

API	*Application Programming Interface.*
Application Acceleration Processor	A processor external to the CPU, used for application acceleration. Commonly refers to a *FPGA.*
Application Specific Integrated Circuit	A chip which is infused with a particular design at production, and cannot be modified afterwards.
ASIC	See *Application Specific Integrated Circuit*.
ChipScope Pro	A logic analyzer from Xilinx, used for live hardware debugging of *FPGA*s using *JTAG* interface ports.
Field-Programmable Gate Array	A general-purpose reconfigurable chip which can be programmed with a virtually arbitrary function.
FPGA	See *Field-Programmable Gate Array*.
GNU	*GNU's Not UNIX*, a set of programs that together with the Linux kernel make up the GNU/Linux OS.
Hardware Description Language	A form of programming language used to describe the behaviour of hardware, for simulation and/or design purposes.
HDL	See *Hardware Description Language.*
HPC	*High Performance Computing.*
IP	*Intellectual Property*, here referring to pre-made modules or cores for ASIC or FPGA circuits.
ISE Foundation	A logic design tool from Xilinx.
JTAG	See *Joint Test Action Group*.
Joint Test Action Group	Developers of IEEE 1149.1-1990, a standard for debugging circuitry and -interface for integrated circuits.
Message Passing Interface	An API for multi-platform parallel programming based on message passing, frequently used for distributed-memory architectures.
ModelSim	A program from Mentor Graphics, used for simulating and debugging *FPGA* designs without actually running them on a *FPGA.*
MPI	See *Message Passing Interface*.
OpenMP	An API for multi-platform shared-memory parallel programming.
QDR II SRAM	*Quad Data Rate Second Generation Static RAM.*
RapidArray Transport	The proprietary Direct Connect interconnect used in the Cray XD1.
RapidArray Processor	A processor used to interface between the *RapidArray Transport* and the CPUs or the FPGA.
RT	See *RapidArray Transport.*
SMP	See *Symmetric Multiprocessing*.
Symmetric Multiprocessing	A multiprocessor system where the processors are identical.
VHDL	*VHSIC Hardware Description Language*.

Bibliography

Note: Electronic documents (PDF files) are included on the accompanying CD-ROM.

[ACTELPWR] Actel, *Power FAQs*: http://www.actel.com/documents/PowerFAQ.pdf

[ALTLIB] Altera Support Library: http://www.altera.com/support/spt-index.html

[AMD1] AMD, *AMD Opteron Processor Model Numbers and Features Comparison*: http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796_9240,00.html

[AMD2] AMD, *Server Application Single-to-Dual Core Scaling*: http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796_8800~97051,00.html

[AMD3] AMD, *AMD64 Architecture Tech Docs*: http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_875_7044,00.html

[AMD3DN] AMD, *3DNow! Technology Manual*, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/21928.pdf

[ASH02] Ashenden, Peter J., *The Designer's Guide to VHDL 2nd Edition*, Morgan Kaufmann, May 2002.

[AT03Q1] arstechnica.com, *An Introduction to 64-bit Computing and x86-64*, http://arstechnica.com/cpu/03q1/x86-64/x86-64-1.html, 2003.

[BROWN04] D.H. Brown, *Cray XD1 Brings High-Bandwidth Supercomputing to the Mid-Market*: http://www.cray.com/downloads/dhbrown_crayxd1_oct2004.pdf, 2004.

[EMVHDL] Emacs VHDL Module: http://opensource.ethz.ch/emacs/vhdl-mode.html, 2005.

[FFAQ05] fpga-faq.org, *FPGA Boards and Systems*: http://www.fpga-faq.org/FPGA_Boards.shtml, 2005.

[GUCC99] Guccione, Steve, *List of FPGA-based Computing Machines*: http://www.io.com/~guccione/HW_list.html, 2000.

[IBMCELL1] Kahle, James A. et al., *Introduction to the Cell Microprocessor*, 2005.

[IBMCELL2] IBM, *Cell Broadband Engine Architecture Version 1.0*, 2005.

[IEEE1149] IEEE, *IEEE Standard Test Access Port and Boundary-Scan Architecture*, IEEE Std 1149.1-1990 & 1149.1a-1993, 1993.

[INTELMMX] http://www.intel.com/support/processors/pentiummmx/

[INTELSSE] http://www.intel.com/support/processors/sb/cs-001650.htm

[KRUT04] Krutådal, Lars & Martisen, May Linda, *The NEC Earth Simulator*, 2004.

[MANO01] M. Morris Mano & Charles R. Kime, *Logic and Computer Design Fundamentals 2nd Edition*, Prentice Hall Inc., 2001.

[MAX04] Maxfield, Clive, *The Design Warrior's Guide to FPGAs*, Newnes Press, 2004.

[OGMMAP] opengroup.org, *The Open Group Base Specifications Issue 6 - mmap*: http://www.opengroup.org/onlinepubs/009695399/functions/mmap.html

[OGOPEN] opengroup.org, *The Open Group Base Specifications Issue 6 - open*: http://www.opengroup.org/onlinepubs/009695399/functions/open.html

[PAT05] David A. Patterson & John L. Hennessy, *Computer Organization and Design 3rd Edition*, Elsevier Inc., 2005.

[PELLE05] Pellerin, David & Thibault, Scott, *Practical FPGA Programming in C*, Prentice Hall Inc., April 2005.

[PGI60UG] The Portland Group/STMicroelectronics, *PGI User's Guide*, Release 6.0, March 2005

[PNR-DD-0015] Cray Inc., *Cray XD1 MINCE FPGA Design*, Issue 0.7, 2005.

[PNR-DD-0022] Cray Inc., *Cray XD1 Mersenne Twister Accelerator FPGA Design*, Issue 1.0, 2005.

[PNR-DD-0023] Cray Inc., *Cray XD1 Hello World FPGA Design*, Issue 0.7, 2005.

[S-2429] Cray Inc., *Cray XD1 System Overview*, Release 1.2, 2005.

[S-2430] Cray Inc., *Cray XD1 System Administration*, Release 1.2.1, 2005.

[S-2433] Cray Inc., *Cray XD1 Programming*, Release 1.2.1, 2005.

[S-2455] Cray Inc., *Cray XD1 1.3.1 Release Notes*, 2005.

[S-6400] Cray Inc., *Cray XD1 FPGA Development*, Release 1.2, 2005.

[S-6411] Cray Inc., *Design of Cray XD1 RapidArray Transport Core*, Release 1.2.1, 2005.

[S-6412] Cray Inc., *Design of Cray XD1 QDR II SRAM Core*, Release 1.2, 2005.
[SC05209] Zhuo, Ling & Prasanna, Viktor K., *High Performance Linear Algebra Operations on Reconfigurable Systems*, Nov 2005.
[SC05311] Tripp, Justin L. et al, *Partitioning Hardware and Software for Reconfigurable Supercomputing Applications: A Case Study*, Nov 2005.
[SGIRASC] SGI, *Reconfigurable Application-Specific Computing User's Guide*, Ver.002, November 2004.
[SMITH01] Smith, Lorna & Bull, Mark, *Development of mixed mode MPI/OpenMP applications*, 2001.
[SUNVIS] SUN, *Vis Instruction Set*, http://www.sun.com/processors/vis/, 2003.
[TRANS05] Tripp, Justin L. et al, *Metropolitan Road Traffic Simulation on FPGAs*, 2005.
[TUTMPI] MPI tutorial: http://www-unix.mcs.anl.gov/mpi/tutorial/
[TUTOMP] OpenMP tutorials: http://www.llnl.gov/computing/tutorials/openMP/
[VHDL87] VHDL87 Syntax: http://opensource.ethz.ch/emacs/vhdl87_syntax.html
[VHDL93] VHDL93 Syntax: http://opensource.ethz.ch/emacs/vhdl93_syntax.html
[WIKI01] wikipedia.org, *AMD64*, http://en.wikipedia.org/wiki/AMD64, 2005.
[WIKI02] wikipedia.org, *CPLD*, http://en.wikipedia.org/wiki/CPLD, 2005.
[WIKI03] wikipedia.org, *CMOS*, http://en.wikipedia.org/wiki/CMOS, 2005.
[WIKI04] wikipedia.org, *Vector processor*, http://en.wikipedia.org/wiki/Vector_processor, 2005
[WIKI05] wikipedia.org, *Cell*, http://en.wikipedia.org/wiki/Cell_(microprocessor) , 2005.
[WIKI06] wikipedia.org, *PlayStation 3*, http://en.wikipedia.org/wiki/PlayStation_3, 2005.
[WIKI07] wikipedia.org, *IEEE 1164*, http://en.wikipedia.org/wiki/IEEE_1164, 2004.
[XCBIO53] Regester, Keith et al, *Implementing Bioinformatics Algorithms on Nallatech-Configurable Multi-FPGA Systems*, 2005.
[XIDL] Xilinx Software Downloads: http://www.xilinx.com/xlnx/xil_sw_updates_home.jsp
[XILIB] Xilinx Support Library: http://www.xilinx.com/support/library.htm
[XIV2PRO] Xilinx, *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, DS083 (v4.5), 2005.
[YALA98] Yalamanchili, Sudhakar, *VHDL Starter's Guide*, Prentice Hall Inc., 1998.

Appendices

A The Cray XD1 System

The XD1 is a recently developed supercomputer system from Cray, based on the idea of taking a conventional system with a fast interconnect, and adding FPGAs as application acceleration processors. This section will give some details of the various components in the system, and how they work together to enable this.

A.1 Technical overview

The base Cray XD1 unit is based on a proprietary RapidArray Interconnection used to connect up to 30 processors of various functions. This interconnection fabric provides both internal and external switching for the RapidArray processors on the compute blades. The basic configuration has 12 internal and external links, which can be expanded to 24 internal and external links with a fabric expansion card.

A base unit holds 6 compute blades, or nodes, with each of these being equipped with dual 64-bit AMD Opteron CPUs, DIMM memory banks (maximum 16GiB/node), and a RapidArray processor for interfacing with the interconnection fabric. Each node runs its own instance of the GNU/Linux operating system. An expansion module can also be attached to each compute blade, providing direct access to a Xilinx Virtex-II Pro FPGA with 4 banks of QDR II SRAM, along with an additional RapidArray processor. One chassis holds 12 CPUs, 6 or 12 RapidArray processors, and 0 or 6 FPGAs.

A PCI-X expansion card provides 3 I/O slots, each of which is connected to two compute blades over dedicated HyperTransport links. These can be used for Gigabit Ethernet or Fibre Channel cards. It also provides an interface to the disk blades. A base unit has one to three disk blades with one or two SATA disk drives per blade, with each disk connected to one compute blade. There are also I/O slots reserved for the use of JTAG interface cards, used for debugging the FPGAs.

The main board also has an independent, specialized AMD AU1000 management processor running the MQX real-time operating system, which monitors the state of the unit, and communicates with the other units over the independent Ethernet-based supervisory network.

Figure A.1 illustrates how all the main components in a base Cray XD1 chassis fit together.

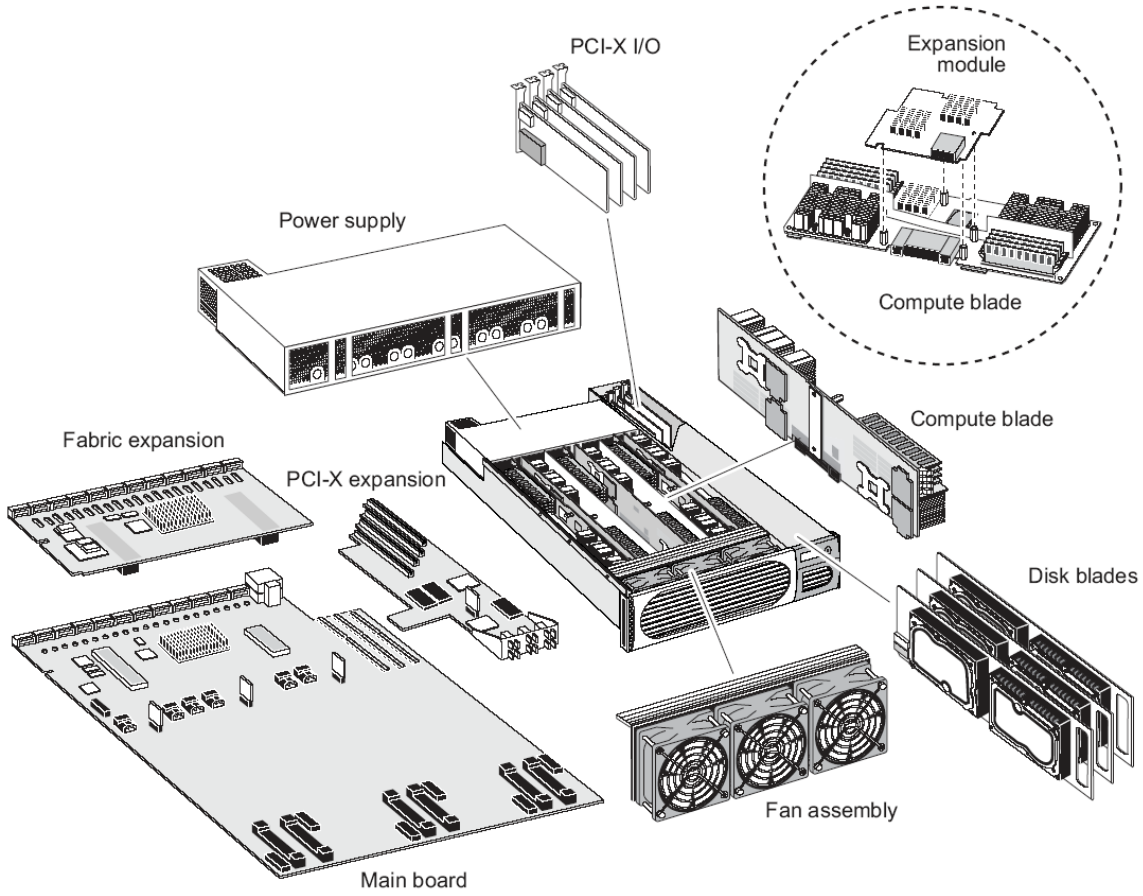

Figure A.1: The main components in a Cray XD1 chassis [S-2429].

A.2 Central components

In this section, certain central parts of the system will be examined in greater detail. Of particular interest is the use of the RapidArray interconnection fabric to move data internally not only between the nodes, but between parts on the nodes themselves. The AMD processors used in the system will be introduced, with some focus on the AMD64 extensions. The expansion modules with the FPGA and connected memory will also be examined. On the software side, certain modifications and additions done to the GNU/Linux operating system for HPC optimization will be discussed.

A.2.1 The RapidArray Interconnect

The basic RapidArray configuration consists of a switch which provides 12 internal links used to connect the internal RapidArray processors situated on the nodes, as well as 12 external links used for chassis interconnection, as illustrated in Figure A.2. The number of internal and external links can be doubled to 24 with the fabric expansion card, which provides an additional RapidArray switch.

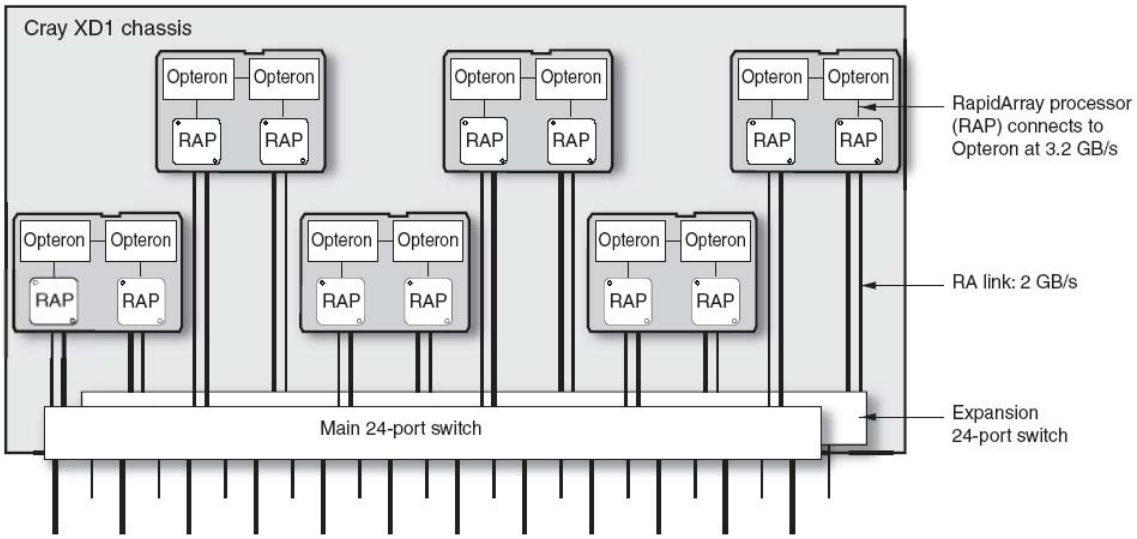

Figure A.2: The RapidArray Interconnect [S-2429].

Each RapidArray link is a unidirectional serial link with a capacity of up to 2GBps. Two links connect to each node using the single fabric, giving 2GBps bidirectional transfer. With the fabric expansion and node expansion modules, four links are used to each node, giving 4GBps bidirectional bandwidth. The maximum aggregate bandwidth per chassis is 96GBps.

The RapidArray interconnect enables processes to directly access user memory on other nodes without going through the Linux kernel. This functionality is used in the implementation of Cray's modified MPICH message-passing library. It is also used to maintain clock synchronization across the entire Cray XD1 system.

The optional FPGAs use the RapidArray interconnect to communicate with the Opteron processors, using a 1.6GBps bidirectional connection to the RapidArray processor. The system is able to transfer data between system memory and the FPGA without interrupting the programs executing on the CPUs. The FPGA interconnection is illustrated in Figure A.3. The FPGA is not able to directly access memory outside of its host node, but it can communicate with neighboring FPGAs in the same chassis through 2GBps RocketI/O links, as indicated in the figure.

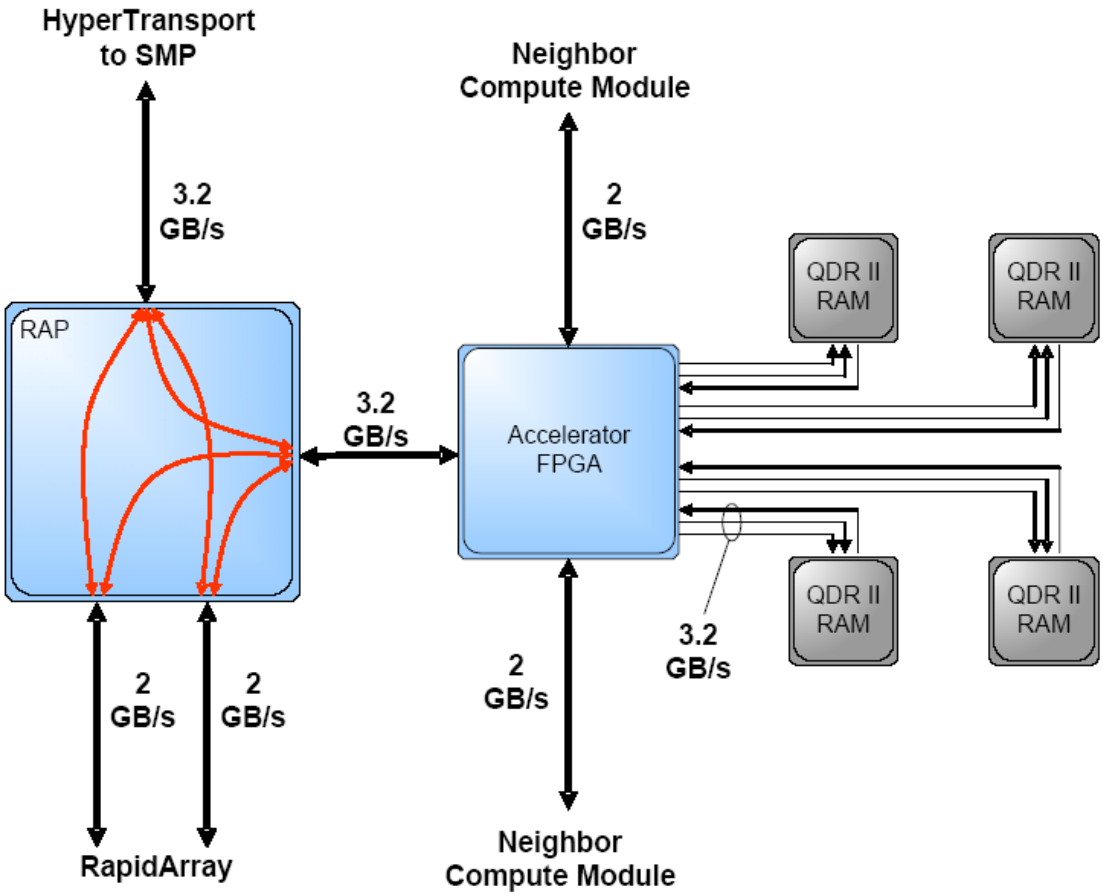

Figure A.3: The FPGA RapidArray Interconnection [S-6400].

A.2.2 The AMD Opteron CPU

The CPUs currently used in the Cray XD1 are AMD Opteron Model 250. These processors come with a single core operating at 2.4GHz, and support up to 2-way SMP configurations. The L1 cache is fully associative, with 64KB for data and 64KB for instructions, while the L2 cache is 4-way set associative, with 1MB shared for data and instructions [AMD1]. These could, if needed, be upgraded to dual-core Model 280 Opteron CPUs, which according to tests made by AMD would make each node about 50-90% faster for practical software use [AMD2].

The major differentiating factor between these CPUs and other x86 CPUs, like Intel's traditional Xeon server line, are the AMD64 (earlier named x86-64) extensions, a set of 64-bit instructions added as an extension to the traditional x86 instruction set. In addition to the obvious, like the ability to directly manipulate 64-bit integer values, this allows implementation to use a much larger memory space. Current implementations allow up to 256 TB, but this can easily be increased in future implementations. Full 64-bit memory addressing would allow a computer to address a memory as large as 2^{64} bytes, or 18 exabytes (18 million terabytes).

Some other important chances include changes in the number of internal registers. The number of General-Purpose integer registers has been increased from 8 to 16, while the size of these registers has been increased from 32 to 64 bits. The number of registers used for SSE has also been increased from 8 to 16. Instructions have been added for relative data addressing, allowing programs to address data relative to the program counter (PC). A so-called NX (*No eXecute)* bit disallows execution of memory areas used for data, which improves security by blocking most forms of buffer overflow attacks, provided software implementations use it [WIKI01].

An introduction to AMD64, intended for people with little or no experience in the hardware field, can be found at [AT03Q1]. Technical design documents for AMD64 development can be found on AMD's website, at [AMD3].

A.2.3 The Xilinx Virtex-II Pro FPGA

The Xilinx Vertex-II Pro FPGA is found on the optional expansion modules, along with four QDR II SRAMs and a programmable clock source for use with the FPGA, as well as an additional RapidArray processor for communication. The FPGA is designed to operate at speeds from 63 to 199MHz, but due to limitations in the QDR II SRAM core it has to be set to operate at speeds from 130 to 199MHz when this is used.

The particular model used in the Cray XD1 is the XC2VP50-7. The Configurable Logic Blocks, which provides combinatorial and synchronous logic, use a total of 53,136 programmable logic cells, with a maximum amount of internally distributed RAM of 738Kb. It has two built-in PowerPC 405 RISC processor blocks, operating at a maximum of 400MHz. 232 Block SelectRAM+ 18Kb memory blocks provide a maximum total of 4,176Kb of storage. Up to 232 Embedded Multiplier Blocks can act as 18 x 18 bit twos-complement signed multipliers, while up to 16 Digital Clock Manager blocks provide a coherent clock signal across the chip, and also provide 90-, 180- and 270-degree phase shifted clock signals. The routing between the blocks use a so-called Active Interconnect Technology to interconnect all elements, using a matrix of routing switches. Full details can be found at [XIV2PRO].

The FPGA is connected to four separate and independent QDR II SRAMs, which can be accessed from FPGA designs using the Cray XD1 QDR II SRAM Core. This core provides an interface for the user part of the FPGA design, and is described in detail in [S-6412]. Each SRAM can store 1M 36-bit words (32 bit data and 4 bit parity), and has independent 36-bit read and write buses, giving a transfer rate of up to 1.6GBps in each direction per SRAM, giving a total maximum bandwidth of 6.4GBps in each direction.

The FPGA is also connected to a RapidArray processor, which can be accessed from FPGA designs using the Cray XD1 RapidArray Transport Core. This core provides an interface for the user part of the FPGA design, and is described in detail in [S-6411]. The interface is used both to initiate communication and process responses for read and write

transaction across the RapidArray fabric, using a 64-bit interface operating at a maximum of 199MHz to the RapidArray processor, providing up to 1.6GBps simultaneous reads and writes. The highest theoretical sustainable rate is about 8/9 of this, or 1.422GBps.
It supports posted writes and multiple outstanding read requests, and can burst up to 64 bytes of data per request.

A.2.4 Modifications to the GNU/Linux Operating System

The operating system used on the Cray XD1 is based on the 64-bit SuSE Linux Enterprise Server 9 distribution, running the Linux kernel version 2.6.5. It includes several enhancements like an improved scheduler and a customized version of MPI. It also has device drivers to access the devices particular to the Cray XD1, such as the FPGA and the RapidArray processor.

The Linux Synchronized Scheduler (LSS) is designed to reduce the impact operating system interrupts have on the overall performance of the system. Certain restrictions on when and how long system services and daemons can run help reduce time wasted on context switches. In addition it synchronizes all nodes within a partition to a common time source, which reduces the time lost when performing collective operations, and makes sure that OS housekeeping tasks are performed at the same time across the nodes. This significantly reduces the impact of so-called “OS jitter”, which occurs when these housekeeping tasks are performed at different times on the various nodes, forcing all the other nodes to wait at synchronization points [BROWN04].

The MPI support in the Cray XD1 is based on MPICH, but is modified to take advantage of the RapidArray interconnection. This allows the MPI calls to interact directly with the RapidArray processor, bypassing the kernel. This reduces time spent on copying data between user- and system memory, and cuts the number of kernel context switches.

Certain other additions, such as the optional Lustre File System and the Active Manager software, are of little concern to the users of the system, and are outside the scope of this text. Information about these can be found at [S-2429] and [S-2430].

B FPGA C API Reference

This appendix contains a reference to the functions provided by the C API, as well as the various error codes used by the API.

B.1 FPGA API Functions

Table B.1 provides a reference to the various functions in the API, including function signature, arguments, return values and a short description. Further information about the API can be found in [S-2433] and the `einlib.h` header file, as well as by typing **`man fpga_intro`** or **`man`** ***`functionname`*** at the XD1 command prompt.

The `einlib.h` header file hints towards future implementations of the API functions `fpga_appmap`, `fpga_appunmap`, `fpga_put`, `fpga_get` and `fpga_intwait`, but the use and intention of these functions is unknown at this time. It also provides an undocumented function `fpga_assert`, which use is not known.

API call signature	Arguments	Returns	Description
`fpga_open``: int` `const char *,` `int,` `err_e *`	*`f_path,`* *`flags,`* *`&err`*	FPGA file descriptor: *`fpga_fd`*.	Opens a file descriptor used to communicate with the FPGA.
`fpga_load``: int` `int,` `const char *,` `err_e *`	*`fpga_fd,`* *`ldfile,`* *`&err`*	Number of bytes uploaded.	Loads the FPGA with the specified loadfile.
`fpga_reset``: int` `int,` `err_e *`	*`fpga_fd,`* *`&err`*	0 on success; -1 on failure.	Places the FPGA in the **reset** state.
`fpga_start``: int` `int,` `err_e *`	*`fpga_fd,`* *`&err`*	0 on success; -1 on failure.	Releases the FPGA from the **reset** state.
`fpga_status``: int` `int,` `err_e *`	*`fpga_fd,`* *`&err`*	FPGA status value in the range 0-255, or -1 on fail.	Retrieves the value of the host latch register in the RA core.
`fpga_is_loaded``: int` `int,` `err_e *`	*`fpga_fd,`* *`&err`*	1 if loaded; 0 otherwise.	Queries whether the FPGA is loaded or not. *
`fpga_unload``: int` `int,` `err_e *`	*`fpga_fd,`* *`&err`*	0 on success; -1 on failure.	Erases any logic programming from the FPGA.

API call signature	Arguments	Returns	Description
fpga_close: int int, err_e *	*fpga_fd,* *&err*	0 on success; -1 on failure.	Closes the FPGA file descriptor.
fpga_memmap: void * int, size_t, int, int, off_t, err_e *	*fpga_fd,* *length,* *protect,* *flags,* *offset,* *&err*	A pointer to the mapped memory in the application address space; NULL on failure.	Maps a region of the FPGA address space at offset *offset* with length *length* to the application address space.
fpga_mem_sync: int int, err_e *	*fpga_fd,* *&err*	0 on success; -1 on failure.	Flushes all outstanding memory transactions.
fpga_wrt_appif_val: int int, unsigned long, unsigned long, unsigned long, err_e *	*fpga_fd,* *value,* *offset,* *type,* *&err*	0 on success; -1 on failure.	Writes the value *value* of type *type* to the FPGA at offset *offset*.
fpga_rd_appif_val: int int, unsigned long *, unsigned long, err_e *	*fpga_fd,* *&value,* *offset,* *&err*	0 on success; -1 on failure.	Reads the value *&value* from the FPGA at offset *offset*.
fpga_set_ftrmem: void * int, unsigned long, err_e *	*fpga_fd,* *order,* *&err*	A pointer to the FPGA transfer region; NULL on fail.	Sets up a FPGA transfer region used for FPGA-initiated reads and writes.

Table B.1: Complete reference of API calls in einlib.h

* Note that this function at the time of this writing had a bug that reversed the results compared to what was stated in the official documentation, but was allegedly corrected in release 1.3.

B.2 FPGA Error Codes

The following error codes are defined by **`typedef enum`** `err_e` in the `einlib.h` header file.

Error Code	Description
NOERR	No error.
FILEOPRERR	File operation system call failure.
INVALIDOP	Invalid API operation requested.
INVALIDVAL	Invalid value passed to the API call.
INVALIDARGS	Invalid argument passed to the API call.
INVALIDINP	Invalid input given to the API call.
DEVOPRERR	FPGA device operation error.
UNKNOWNERR	Unknown error.

Table B.2: Error code definitions in `einlib.h`

C Cray XD1 stresstest and problems

During the course of writing this paper, several problems were encountered with the Cray XD1 hardware (some API problems are covered in Section 5.4.2). These problems would manifest themselves as random lockups, or output that differed from the results given in the simulations. Worse yet, after some time and many hours of fruitless debugging it became evident that some of the problems also manifested when using the Cray sample designs! A comprehensive test was done using the MINCE test, described in Section 7.5.2, and the results are given below.

Of course, it is likely that some (or most) of the problems encountered while testing the Cray XD1 are due to faulty user designs, but the results of the stresstest show that at least some of the problems are likely to be with the Cray XD1 itself. Enough information is provided to allow others to repeat the test with the same setup.

C.1 System Information

The following is the system data reported by `lsnode –verbose`. The data here reflects the system setup, including software build versions and system-specific information about the FPGA. Only the nodes used in the stresstest, namely 403.2, 403.3 and 403.5 are included. Note that the bundle version on the nodes is 1.2, and while a newer release is available, and could possibly fix the problems, this was not in place before this text was finalized. The release notes for 1.3/1.3.1 [S-2455] does however not mention any bugs that are likely to have caused this.

```
Hardware ID:                  403.2
Partition:                    compute
Access Control:               open
Compute Blade Bundle Version: 1.2build1020
State:                        online-open
Status Message:               --
Critical Alarms:              0
Major Alarms:                 0
Minor Alarms:                 0
Warning Alarms:               0
WC Storage Location:          local-disk (/dev/hda2)
Hostname:                     musculus403-2
IP Addresses:
    10.128.25.50
    10.0.25.50
Services:
Hosted NIC Ports:
Terminated NIC Ports:
App Accelerator:              87-0003-11
Kernel Version:               2.6.5_H_01_02
Kernel Location:              --
Boot Parameters:              --
Restart on Failure:           --
```

```
Memory Total(KB):             4114004
Memory Used(KB):              435524
Memory Free(KB):              3678480
Swap Total(KB):               5245180
Swap Used(KB):                0
Swap Free(KB):                5245180
Load Avg 1:                   1.0
Load Avg 5:                   1.0
Load Avg 15:                  1.0
CPU List:
    Model    cpu #0: AMD Opteron(tm) Processor 250 cpu #1: AMD
Opteron(tm) Processor 250
    MHz      cpu #0: 2393.658 cpu #1: 2393.658
    Cache    cpu #0: 1024 KB cpu #1: 1024 KB
    Temp(C)  cpu #0: 45.27   cpu #1: 46.66

Hardware ID:                  403.3
Partition:                    compute
Access Control:               open
Compute Blade Bundle Version: 1.2build1020
State:                        online-open
Status Message:               --
Critical Alarms:              0
Major Alarms:                 0
Minor Alarms:                 0
Warning Alarms:               5
WC Storage Location:          local-disk (/dev/hda2)
Hostname:                     musculus403-3
IP Addresses:
    10.128.25.51
    10.0.25.51
Services:
Hosted NIC Ports:
Terminated NIC Ports:
App Accelerator:              87-0003-11
Kernel Version:               2.6.5_H_01_02
Kernel Location:              --
Boot Parameters:              --
Restart on Failure:           --
Memory Total(KB):             4114004
Memory Used(KB):              151784
Memory Free(KB):              3962220
Swap Total(KB):               5245180
Swap Used(KB):                0
Swap Free(KB):                5245180
Load Avg 1:                   1.0
Load Avg 5:                   0.99
Load Avg 15:                  0.72
CPU List:
    Model    cpu #0: AMD Opteron(tm) Processor 250 cpu #1: AMD
Opteron(tm) Processor 250
    MHz      cpu #0: 2393.643 cpu #1: 2393.643
    Cache    cpu #0: 1024 KB cpu #1: 1024 KB
    Temp(C)  cpu #0: 44.92   cpu #1: 46.14

Hardware ID:                  403.5
Partition:                    compute
```

```
Access Control:               open
Compute Blade Bundle Version: 1.2build1020
State:                        online-open
Status Message:               --
Critical Alarms:              0
Major Alarms:                 0
Minor Alarms:                 0
Warning Alarms:               8
WC Storage Location:          local-disk (/dev/hda2)
Hostname:                     musculus403-5
IP Addresses:
    10.128.25.53
    10.0.25.53
Services:
Hosted NIC Ports:
Terminated NIC Ports:
App Accelerator:              87-0003-11
Kernel Version:               2.6.5_H_01_02
Kernel Location:              --
Boot Parameters:              --
Restart on Failure:           --
Memory Total(KB):             4114004
Memory Used(KB):              155272
Memory Free(KB):              3958732
Swap Total(KB):               5245180
Swap Used(KB):                0
Swap Free(KB):                5245180
Load Avg 1:                   1.0
Load Avg 5:                   1.0
Load Avg 15:                  0.84
CPU List:
    Model    cpu #0: AMD Opteron(tm) Processor 250 cpu #1: AMD
Opteron(tm) Processor 250
    MHz      cpu #0: 2393.658 cpu #1: 2393.658
    Cache    cpu #0: 1024 KB cpu #1: 1024 KB
    Temp(C)  cpu #0: 45.68   cpu #1: 42.69
```

C.2 Test Setup

The test here is a sanity testing using the MINCE tool, which as mentioned is one of the example programs provided by Cray. The Cray-provided `test.sh` script, repeated at the end of this section, invokes the tests using a variety of settings. In all cases, the C files were unaltered, and generated by the accompanying Makefile.

Tests 1 and 3 were run using the binary provided by Cray, located at `/opt/ufpapps/mince/bin/mince_xc2vp50.bin`.

Tests 2 and 4 were run using a binary generated by the Windows version of Xilinx ISE 7.1_04i from the files located at `/opt/ufpapps/mince/src/80-0008_mince`.

Test 1 and 2 runs the designs at the speed Cray suggests, namely 190MHz. The speed was then dropped to 140MHz, to see if this would mitigate the problems observed. The tests were done using the ufp header files given at the end of this section.

Note that Test 1 and 2 are mostly run on only two of the test nodes. The third node was brought into action only halfway through the test, and even though it was the intention to run Test 1 and 2 on this node as well, this was postponed due to the temperature problems mentioned in Section C.4.

`test.sh`:

```
./berttest  -v -r -w 0 -t 60 -d rand -a incr
./berttest  -v -r -w 0 -t 60 -d slam -a incr
./berttest  -v -r -w 0 -t 60 -d incr -a incr
./berttest  -v -r -w 0 -t 60 -d rand -a rand
./berttest  -v -r -w 0 -t 60 -d slam -a rand
./berttest  -v -r -w 0 -t 60 -d incr -a rand

./berttest  -v -r -w 1 -t 60 -d rand -a incr
./berttest  -v -r -w 1 -t 60 -d slam -a incr
./berttest  -v -r -w 1 -t 60 -d incr -a incr
./berttest  -v -r -w 1 -t 60 -d rand -a rand
./berttest  -v -r -w 1 -t 60 -d slam -a rand
./berttest  -v -r -w 1 -t 60 -d incr -a rand

./berttest  -v -r -w 2 -t 60 -d rand -a incr
./berttest  -v -r -w 2 -t 60 -d slam -a incr
./berttest  -v -r -w 2 -t 60 -d incr -a incr
./berttest  -v -r -w 2 -t 60 -d rand -a rand
./berttest  -v -r -w 2 -t 60 -d slam -a rand
./berttest  -v -r -w 2 -t 60 -d incr -a rand

./berttest  -v -r -w 3 -t 60 -d rand -a incr
./berttest  -v -r -w 3 -t 60 -d slam -a incr
./berttest  -v -r -w 3 -t 60 -d incr -a incr
./berttest  -v -r -w 3 -t 60 -d rand -a rand
./berttest  -v -r -w 3 -t 60 -d slam -a rand
./berttest  -v -r -w 3 -t 60 -d incr -a rand

./qdrtest -v
./qdrtest -v -i
./qdrtest -v -d rand
```

`ufphdr` for Test 1 and 2:

```
Cray Part Number   : 90-0003-08;
FPGA Frequency MHz : 190;
```

`ufphdr` for Test 3 and 4:

```
Cray Part Number   : 90-0003-08;
FPGA Frequency MHz : 140;
```

C.3 Test Results

Test 1: Cray binary @190MHz

403-3 Run 1: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 1 -t 60 -d incr -a incr
403-3 Run 2: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 2 -t 60 -d incr -a incr
403-3 Run 3: Success
403-3 Run 4: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 0 -t 60 -d slam -a incr

403-5 Run 1: Success
403-5 Run 2: Failure - Error message reproduced at (Error 1) below. Persistent error, manually rebooted.
403-5 Run 3: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 0 -t 60 -d rand -a incr
403-5 Run 4: Success

Test 2: Xilinx ISE 7.1_04i binary @190MHz

403-2 Run 1: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 0 -t 60 -d incr -a incr

403-3 Run 1: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 1 -t 60 -d incr -a rand
403-3 Run 2: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 0 -t 60 -d slam -a incr
403-3 Run 3: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 1 -t 60 -d rand -a rand
403-3 Run 4: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 1 -t 60 -d incr -a rand

403-5 Run 1: Success
403-5 Run 2: Failure - Error message reproduced at (Error 1) below. Persistent error, manually rebooted.
403-5 Run 3: Success
403-5 Run 4: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 0 -t 60 -d slam -a incr

Test 3: Cray binary @140MHz

403-2 Run 1: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 0 -t 60 -d slam -a rand
403-2 Run 2: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 1 -t 60 -d slam -a rand
403-2 Run 3: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 1 -t 60 -d incr -a incr
403-2 Run 4: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 1 -t 60 -d slam -a incr

403-3 Run 1: Success
403-3 Run 2: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 3 -t 60 -d rand -a incr
403-3 Run 3: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 0 -t 60 -d slam -a rand
403-3 Run 4: Success

403-5 Run 1: Success
403-5 Run 2: Success
403-5 Run 3: Success
403-5 Run 4: Success

Test 4: Xilinx ISE 7.1_04i binary @140MHz

403-2 Run 1: Success
403-2 Run 2: Failure - Error message reproduced at (Error 1) below. Persistent error, manually rebooted.

403-2 Run 3: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 2 -t 60 -d slam -a incr
403-2 Run 4: Success

403-3 Run 1: Failure - Node locked up and rebooted at test: ./berttest -v -r -w 3 -t 60 -d slam -a incr
403-3 Run 2: Aborted (high temperature alert)
403-3 Run 3: -
403-3 Run 4: -

403-5 Run 1: Success
403-5 Run 2: Failure - Error message reproduced at (Error 1) below. Persistent error, manually rebooted.
403-5 Run 3: Success
403-5 Run 4: Success

(Error-1)

This exact error appeared a number of times during the test runs.

```
Test Parameters -
      [Various Parameters]

Loading file mince_xc2vp50.bin.ufp onto the FPGA.
Executing tests.
*** SMP detected error in memory at RAM offset: 0x15.
    Found:    0x00000000DEADBEEF.
    Expected: 0x0000000000000015.
*** SMP detected error in memory at RAM offset: 0x16.
    Found:    0x00000000DEADBEEF.
    Expected: 0x0000000000000016.
*** SMP detected error in memory at RAM offset: 0x17.
    Found:    0x00000000DEADBEEF.
    Expected: 0x0000000000000017.
*** SMP detected error in memory at RAM offset: 0x18.
    Found:    0x00000000DEADBEEF.
    Expected: 0x0000000000000018.
*** SMP detected error in memory at RAM offset: 0x19.
    Found:    0x00000000DEADBEEF.
    Expected: 0x0000000000000019.
Quitting at 5 errors.
Test FAILED.
```

C.4 Test Evaluation

Node lockups were observed relatively common while using the MINCE tool provided by Cray, both when using the provided binary and when generating binaries using Xilinx ISE 7.1_04i. Tests were run at 190MHz (Cray default) and 140MHz, and while it looks like the success rate is higher at 140MHz, with node 403-5 as an example failing 4/8 tests at 190MHz and only 1/8 on 140MHz, there is not enough data to be able to say anything conclusively. Since the QDR II SRAM core demands a clock rate of at least 130MHz, it is not possible to reduce it much further.

Another error is occasionally reported, where it seems like the writes issued from the FPGA fail for some unknown reason.

Yet another concern is the high temperatures reported towards the end of this test. According to the measurement tools, temperatures on the FPGA on node 403-3 reached as high as 127C, enough to trigger several warnings from the monitoring system. The test was run concurrently on the three nodes, but if this is sufficient to generate dangerously high temperatures, it is possible that the chassis does not have enough cooling to safely utilize all the nodes concurrently. This will of course depend on the application running on the FPGA.

It is possible that the high temperature is the culprit for the high number of failures encountered during this test, but as the results from this test has been reported to Cray, a response had not been given at the time this text was finalized, so this remains pure speculation.