High Performance Scientific Computing Using FPGAs for Lattice QCD

Owen Callanan

A thesis submitted to the University of Dublin, Trinity College in fulfilment of the requirements for the degree of Doctor of Philosophy (Computer Science)

October 2006

Declaration

I, the undersigned, declare that this work has not been previously submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

Owen Callanan

Dated: 31 October 2006

Permission to Lend and/or Copy

Summary

The recent development of large FPGAs combined with the availability of a variety of FPGA-based non-integer arithmetic cores has made it possible to implement high-performance matrix kernel operations on FPGAs. This thesis seeks to evaluate the performance of FPGAs for real scientific computations by implementing lattice Quantum Chromodynamics (lattice QCD), which is one of the classic scientific computing problems. Lattice QCD computing machinery is the focus of considerable research work worldwide, including two custom ASIC based solutions and a variety of custom built PC cluster machines. This wide variety of highly optimised lattice QCD computing machinery permits comparison with the state of the art for high performance computing machinery.

The results presented in this thesis give significant insights into the usefulness of FPGAs for scientific computing. This thesis also evaluates two different number systems available for running scientific computing applications on FPGAs. FPGA based lattice QCD processors are implemented using both double precision IEEE floating point and logarithmic arithmetic cores with precision equivalent to IEEE single precision floating point. The performance of the FPGA based lattice QCD processors is compared with that of two lattice QCD targeted custom ASIC based supercomputers, with that of commercial supercomputers and with that of some highly optimised PC cluster based machines.

The logarithmic arithmetic designs return per FPGA performance of 1320 MFLOPS for the performance critical lattice QCD Dirac operator, and they return 1050 MFLOPS for the full conjugate gradient solver application. The latest generation of PC clusters return per processor performance of about 1300 MFLOPS for the Dirac operator using single precision arithmetic. Thus the logarithmic arithmetic designs are competitive with the latest PC cluster machines, which are the main platform for single precision lattice QCD calculations.

The double precision designs return performance of 1200 MFLOPS for the core

Dirac operator and 940 MFLOPS for the conjugate gradient solver application. This compares very well with the double-precision per-processor performance of the latest PC clusters at 650 MFLOPS and with the performance of the IBM BlueGene/L supercomputer at 1100 MFLOPS per processor. BlueGene/L processors consist of an ASIC with two CPU cores, so the per-core performance of the BlueGene/L is 550 MFLOPS. The double precision FPGA design's performance also compares very well with the per-processor performance of the two custom ASIC supercomputers that have been constructed specifically for lattice QCD. The QCDOC machine has per-processor performance of 396 MFLOPS, whilst the apeNEXT system has per-processor performance of 894 MFLOPS. All figures are for the Dirac operator.

All current lattice QCD machines are constructed using many processors. The computational requirements of lattice QCD are so great that they can never be met by a single processor. To investigate the viability of multiple-FPGA based systems, a dual FPGA version of the Dirac operator was implemented. Lattice QCD is a highly parallelisable problem and can be implemented efficiently on multiple processor machines. The dual-FPGA Dirac operator, which is based on the logarithmic Dirac operator, uses a low latency communications system to allow two FPGAs to work together on a single application of the Dirac operator. A speedup of 1.98 times is delivered over the single FPGA design, by parallelising computation and calculation in the dual FPGA design. This result strongly indicates that FPGAs have the potential to form a scalable multiple processor platform for high performance computing applications such as lattice QCD.

These three sets of results demonstrate that FPGAs can return excellent performance for a typical high performance computing application, lattice QCD, using two different arithmetic systems. Double precision floating point is the most commonly used arithmetic system for high performance computing applications. This makes the results from the double precision designs particularly significant since they demonstrate that FPGAs can return highly competitive performance for real scientific computing applications using double precision arithmetic. Finally the dual-FPGA Dirac operator demonstrates that FPGAs have the potential to form a scalable multiple processor platform for high performance computing.

Acknowledgements

Many people helped me complete this thesis, but no one did more than my supervisor David Gregg. His advice and guidance was vital to the completion of this work, and for this I owe him many thanks.

I would also like to thank Mike Peardon and Jim Sexton for their invaluable expert advice and information on lattice QCD and for providing the algorithms that provided the basis of this work. Thanks are also due to my initial supervisor, Andy Nisbet, for seeing me through the first 2 years of my PhD.

I shared an office with several people over the last four years, and, along the way, they helped me in many ways with my research. I would like to thank Emre Ozër for helping me greatly in my initial efforts at publication, and also Milan Tichy for giving me the benefit of his considerable FPGA design expertise and knowledge. Their assistance was invaluable. I would also like to thank Ross Brennan and Eoin Creedon for many interesting discussions and for helping me solve many tricky problems.

I could not have started this work without the support of my parents and family, and I certainly would never have finished it. Their support and patience helped my through many obstacles and kept me going through difficult times. Simple thanks are not enough, but they will have to suffice (at least for now).

Finally, above all, thanks to Eadaoin Boyle-Tobin, for everything.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Scientific computing has an insatiable appetite for processing power. For several decades computers have been used to investigate natural phenomena such as fluid dynamics, particle simulations and even the weather. By simulating these phenomena using computer simulations we can gain a greater understanding of how the underlying systems work, and predict the future behaviour of these systems.

The use of computers in science is not restricted to simulating the known behaviour of natural phenomena. A significant branch of scientific computing has developed where computer simulations are used to test mathematically expressed scientific theories that are too complex to be proven by solving the equations of the theory, and which cannot be proven by experimental observation.

This technique is used heavily in a particular branch of sub-atomic particle physics theory called quantum chromodynamics, or QCD. QCD theory attempts to explain the behaviour of sub-atomic particles called quarks, which are the basic components of particles like protons and neutrons. However quarks can never actually be observed on their own as they are always bound together inside a larger particle. This makes QCD theory impossible to verify through experimental observation alone.

As a result, physicists have turned to using computer simulations to simulate their theories about how quarks behave. By comparing the results of these simulations to the results obtained from experiments conducted using large particle colliders, such as those at CERN and Fermilab, physicists can test QCD theory. This use of computer simulations to test QCD theory is commonly referred to as lattice QCD.

The computational requirements of these lattice QCD simulations are very large. The latest generation of dedicated lattice QCD computing machines have sustained performance for a single problem of several Teraflops (10^{12} floating point operations per second). With these machines physicists are able to test QCD theory more rigorously than ever before. Yet even these machines are not enough; the lattice QCD community is already discussing how to build Petaflop (10^{15} floating point operations per second) scale machines.

Lattice QCD is only just one of many scientific computing applications which have huge computational requirements, but it provides an excellent example application for testing the performance of a new computing platform for scientific computing. In common with many scientific computing applications, it is matrix based, and is highly parallelisable. Also a significant amount of research effort has been spent on developing dedicated lattice QCD computing machinery, including two custom-ASIC based supercomputers [Belletti '06][P A. Boyle '05]. Commercial supercomputers and PC clusters are also used extensively for lattice QCD. These lattice QCD machines represent the state of the art of scientific computing machinery.

1.1 FPGAs for Scientific Computing

FPGAs, or field programmable gate arrays, are dynamically reprogrammable microchips that can be programmed to take on the behaviour of any digital circuit. They can be used to implement processors that are customised to the specific needs of an application, without the prohibitive expense and design complexity of producing a custom application specific integrated circuit (ASIC). FPGA based processors have been used successfully for some time for integer and fixed-point arithmetic domain applications such as digital signal processing. Designers are able to exploit the inherent parallelism available on an FPGA to implement processors that return excellent performance for these applications.

Historically FPGAs have not been large enough to support a significant number of floating arithmetic units on a single FPGA, which has limited the use of FPGAs for scientific computing. However recent multi-million gate equivalent FPGAs make it possible to implement complex, high-performance designs incorporating floating point arithmetic. Investigating the use of FPGAs for high performance scientific computing is now a significant branch of reconfigurable computing research [Underwood '04b][Zhuo '04].

This thesis investigates the performance of FPGAs for *full* floating-point based scientific applications by implementing the lattice QCD Dirac operator, which is the performance critical part of nearly all lattice QCD simulations. The operator is used to implement a full lattice QCD conjugate gradient solver. The performance results of these implementations are compared with the performance of a number of platforms currently used for lattice QCD simulations.

1.2 Lattice QCD & Lattice QCD Machines

Improving the performance of lattice QCD is the focus of considerable research work worldwide [Wettig '05], including two competing custom ASIC supercomputers targeted solely at lattice QCD simulations [Belletti '06][P A. Boyle '05]. Several groups are also investigating optimizing PC clusters for lattice QCD, [Gellrich '03][Holmgren '05a][Holmgren '05b][Holmgren '06]. Lattice QCD has also been implemented on the latest generation of commercial supercomputers including the IBM BlueGene/L and the SGI Altix machines [Wettig '05]. This considerable body of research allows a comparison of the performance of the FPGA based solution with the state of the art for scientific computing.

The core of lattice QCD is the *Dirac operator*, a large complex floating-point intensive matrix computation. Although the performance of the Dirac operator is central, lattice QCD involves other operations which can have a significant impact on performance. The impact of these operations is investigated by implementing a full example application, a lattice QCD conjugate gradient solver which uses the Dirac operator.

The computational requirements of lattice QCD are too great to be met by a single processor. Machines used for lattice QCD consist of many processors connected together to work on the same problem. For example one QCDOC machine, an example of the custom ASIC supercomputers mentioned previously, consists of 12,288 interconnected custom ASIC processors, which can all work together on the same problem. Such a machine can sustain over five teraflops (10^{12} floating point operations per second) on actual lattice QCD simulations. Consequently a single FPGA will never be sufficient to meet the demands of a full lattice QCD simulation. I have implemented a dual-FPGA version of the core Dirac operator to investigate whether multiple FPGA based systems are good for lattice QCD.

1.3 Arithmetic Systems

This thesis also aims to compare the suitability of different non-integer arithmetic systems for high performance computing applications on FPGAs. A comparison of logarithmic arithmetic and IEEE double precision floating point is presented. This comparison is used to identify the strengths and weaknesses of both arithmetic systems when used for real high performance scientific computing applications. Logarithmic arithmetic is very different to conventional IEEE floating point, and thus has very different constraints when used for real applications.

Performance results are presented for both log arithmetic and IEEE double precision arithmetic versions of the full conjugate gradient solver along with results for the component parts of the conjugate gradient application. These results include performance data for the performance critical Dirac operation. A quantitative comparison is also presented with three categories of lattice QCD systems: ASIC-based processors designed exclusively for lattice QCD, highly optimised PC cluster systems and finally commercial supercomputers. Finally results are presented for a dual-FPGA Dirac operator. These results show that parallelisation of calculation and inter-FPGA communication can maximise performance for multiple-FPGA systems.

1.4 Contributions

This thesis evaluates the viability of FPGAs for real scientific computing applications by implementing FPGA designs that are customised to the needs of a specific, highly demanding, application, lattice QCD.

- *Performance of FPGAs for Scientific Computing*

  The performance of FPGAs is evaluated for a typical, full scientific computing application, lattice QCD. Lattice QCD computing machinery is the focus of a substantial amount of research effort worldwide. By comparing the results obtained from the work described in this thesis with this body of research I am able to give significant insights into the performance of FPGAs for scientific computing.

- *Arithmetic Systems for FPGAs*

  In this thesis performance critical lattice QCD algorithms are implemented for both single precision equivalent logarithmic arithmetic and IEEE compliant double precision floating point. The results of these implementations show the advantages and disadvantages of each arithmetic system when applied to scientific computing applications. Also the double precision implementations are amongst the first FPGA-based implementations of double precision scientific computing applications whilst the logarithmic arithmetic designs are the first successful use of log arithmetic for scientific computing.

- *Multiple FPGA processing for scientific computing*

  The final stage of this work implements a dual FPGA version of the performance critical Dirac operator. This work demonstrates that two FPGAs can work together on the same problem, and can return excellent performance. The result shows that FPGAs have the potential to be used as the basis for highly parallel scientific computing machines that consist of multiple FPGAs.

1.5 Publications Related to this Thesis

The work presented in this thesis has been published in IEEE sponsored international conferences. The implementation of the logarithmic arithmetic Dirac operator, described in Chapter 5, was previously presented at the 12th Reconfigurable Architectures Workshop, which was held as part of the 19th IEEE International Symposium on Parallel and Distributed Processing in 2005 [Callanan '05]. The work described in this paper is the authors own work. The other authors contributed to the paper in the following ways: Jim Sexton provided the original lattice QCD algorithm along with information and expertise on lattice QCD, Emre Ozër aided the work at the design stages and played a significant role in writing the final paper, whilst Andy Nisbet and David Gregg both supervised my PhD studies. Andy Nisbet was my original supervisor when this work commenced, however after Andy Nisbet left his position in Trinity College, David Gregg took over as supervisor.

The double precision Dirac operator and conjugate gradient solvers detailed in Chapter 6 were presented at the IEEE sponsored 16th International Conference on Field Programmable Logic and Applications in 2006 [Callanan '06]. The logarithmic arithmetic conjugate gradient solver, along with an improved version of the logarithmic arithmetic Dirac operator, was also presented at this conference. The work described in this paper is the authors own. The contributions of the credited authors are as follows: Mike Peardon provided the conjugate gradient application along with lattice QCD expertise, and once again David Gregg and Andy Nisbet are credited for their roles in supervising the PhD work.

1.6 Guide to this Thesis

Chapter 2

This chapter presents background information on the systems and technologies used in the work. This information is presented to aid understanding of the work performed for this thesis. An overview of FPGA technology is presented, together with a description of the FPGA development boards used to implement the designs I created for this thesis. Also presented are details of the two arithmetic systems used, logarithmic arithmetic and IEEE floating point arithmetic.

Chapter 3

A comprehensive review of the literature is contained in this chapter, starting with a discussion of what quantum chromodynamics (QCD) theory attempts to explain and why computer simulations must be used to test this theory. The application of computer simulations to test QCD theory is called lattice QCD. A number of articles, written for a general audience by QCD experts, are used to help with this explanation.

A survey of the state of the art of lattice QCD machinery is then presented. Three categories of machine are used for lattice QCD simulations; custom ASIC based supercomputers, commercial supercomputers and PC clusters. All three categories are discussed in detail.

The chapter concludes with a detailed history of the two arithmetic systems used in this work, combined with a review of recent FPGA research into both systems. The first system reviewed is IEEE double precision floating point arithmetic. The history of floating point arithmetic is presented together with a description of the IEEE floating point standard. The development of FPGA based floating point arithmetic is then presented, up to the most recent investigations of common scientific computing kernels for FPGAs using double precision arithmetic.

The second arithmetic system reviewed is logarithmic arithmetic, which is used as an alternative arithmetic system in this work. The concept of logarithmic arithmetic has existed for some time; however it is only now with the availability of FPGA implementations of the system that it is viable for use on real applications. The history of the logarithmic arithmetic system is described, up to the latest FPGA implementation of the logarithmic arithmetic system that is used in this work.

Chapter 4

A detailed algorithm analysis is the subject of this chapter. The analysis is presented separately since it is relevant to all three implementation chapters that follow. The core lattice QCD Dirac operator is analysed in detail, opportunities for parallelisation are identified, and the memory bandwidth requirements of the operator are described. The analysis is then repeated for the conjugate gradient solver application, and for the extra operations that it requires. This analysis formed a vital input to the designs presented in the next three chapters.

Chapter 5

The design and implementation of the first of the three main components of this thesis is discussed in this chapter. This part of the work consists of an FPGA based Dirac operator implementation which uses logarithmic arithmetic for all non-integer arithmetic. This work was then used to produce a full lattice QCD conjugate gradient solver on an FPGA. These implementations are the first use of logarithmic arithmetic on FPGAs for high performance computing problems. Consequently the performance of these designs offers significant insight into the suitability of FPGA based logarithmic arithmetic for scientific computing.

Chapter 6

The initial stage of this work, detailed in Chapter 5 used logarithmic arithmetic to handle all non-integer arithmetic. Logarithmic arithmetic was partly chosen because of the limited availability of conventional IEEE floating point arithmetic cores at the start of the project. However during the project double precision floating point cores became available for FPGAs. Users of lattice QCD prefer double precision arithmetic since it gives a more accurate result, however single precision arithmetic can be used, with some algorithmic adjustments to compensate for the lower precision.

Double precision is the standard arithmetic system for scientific computing, single precision does not have sufficient range or precision for many scientific computing applications. Thus the results of an investigation of double precision floating point are applicable to the full range of scientific computing applications. To investigate the performance of double precision floating point arithmetic on FPGAs, double precision versions of the lattice QCD algorithms were implemented.

The designs detailed in this chapter provide important insights into the performance of double precision floating point arithmetic on FPGAs when used for complete applications. The prior work on double precision floating point arithmetic on FPGAs has been primarily concerned with matrix kernel operations, and there has been little work on full applications.

Chapter 7

Lattice QCD calculations have massive computational requirements which cannot be met by single processor machines. However lattice QCD simulations are highly parallelisable and can be run successfully on massively parallel machines. All machines currently used for lattice QCD simulations are highly parallel multiple processor machines, which can use many processors on a single calculation. It is highly unlikely that a single FPGA lattice QCD machine will ever have sufficient computational power for full lattice QCD simulations.

This chapter details the design and implementation of a bi-directional FPGA communication system, and its use to implement a dual FPGA version of the logarithmic arithmetic Dirac operator that was presented in Chapter 5. This dual FPGA Dirac operator has nearly twice the performance of the single FPGA version. The performance critical Dirac operator takes up over 90% of the time in lattice QCD simulations and requires only nearest neighbour communications when implemented on multiple processor machines. Consequently the results for this part of the work demonstrated that FPGAs have the potential to be a scalable platform for lattice QCD simulations

Chapter 8

This chapter presents performance results for the three parts of this work and compares these results with the performance of each of the three categories of machines that are used for lattice QCD simulations, which are described in Chapter 3. Much research effort is devoted to developing computing machinery for lattice QCD simulations and, as a result, these machines can be taken to represent the state of the art for scientific computing machinery. By comparing the performance of my FPGA based designs with these systems I am able to offer significant insights into the performance of FPGAs for scientific computing applications.

Chapter 9

In the last chapter a number of issues raised by the results presented in this thesis are discussed, and the thesis concludes with a summary of the contributions of my work, and together with an outline of some interesting areas for future work.

Chapter 2

Technologies

This chapter presents background information on the technologies and tools that were used to complete this work. A brief overview of FPGAs and the specific architectural features of the FPGAs used in this work are presented. This is followed by a description of the FPGA development platforms used to implement the designs in hardware. The designs in this thesis were implemented using Handel-C which is a hardware design language that offers a higher level of abstraction than existing FPGA design tools such as VHDL or Verilog. The development and history of Handel-C is discussed, along with its advantages over more conventional hardware design tools. Some basic examples of Handel-C code are included to show how Handel-C is used for FPGA design. Finally the two packages of non-integer arithmetic cores that are used are discussed, and they are compared with examples of equivalent floating point units that are taken from the literature.

2.1 FPGAs

FPGA stands for field programmable gate array. FPGAs are a category of programmable logic devices, or PLDs, that can be programmed in the field (hence the name) after manufacture. The devices themselves consist of an array of logic cells which can be programmed and connected together to implement complex digital circuits in hardware. There are a number of different types of logic cell on current FPGAs. They are:

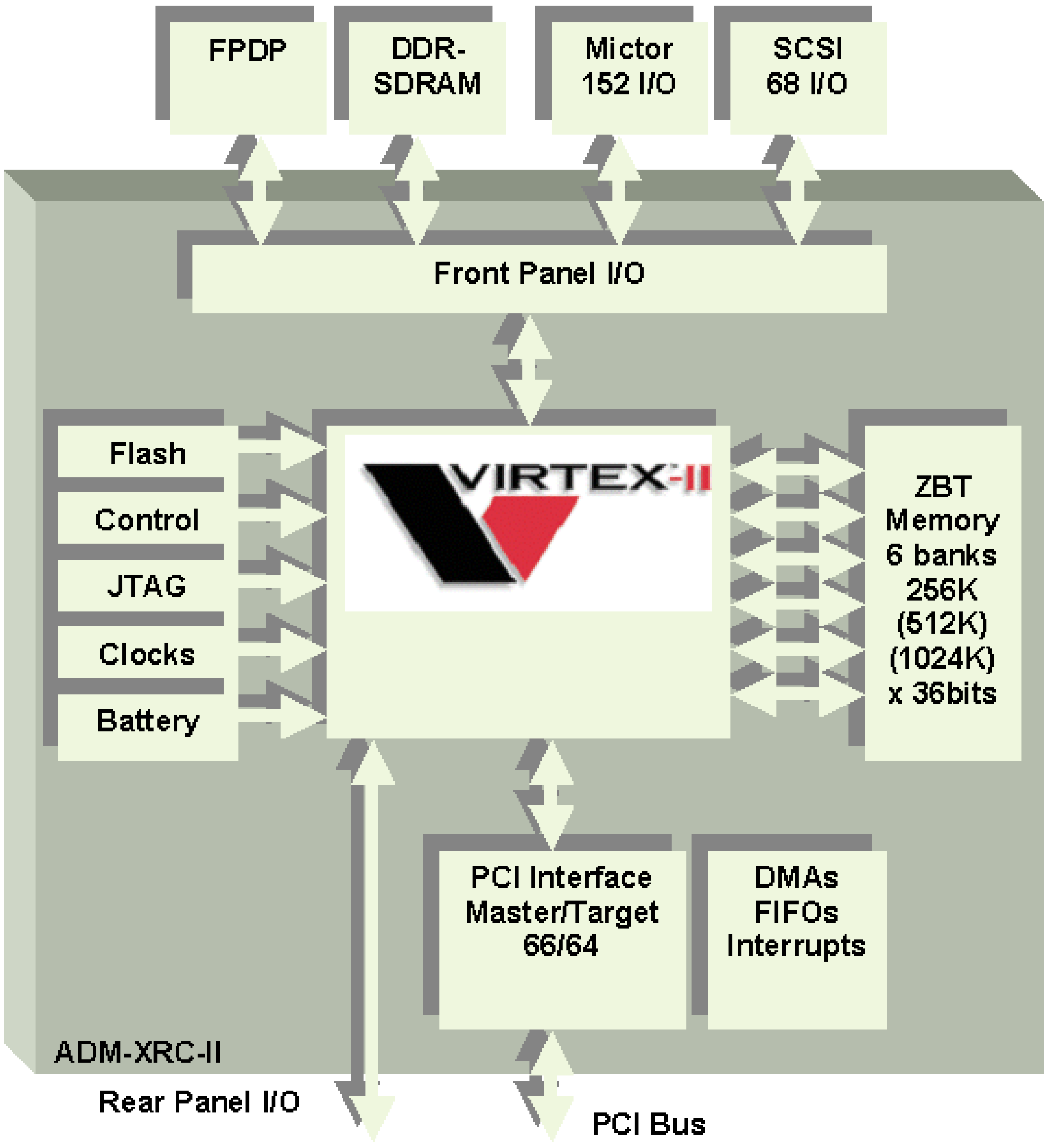


Figure 2-a. Logical diagram of the ADM-XRC-II development board showing the connection of the Virtex-II FPGA to the PCI bus, off-ship SRAM memory and other off-chip components [Alpha-Data '05]

- Slices (containing two flip flops and two small look up tables)

- Block RAMs

- Hardware Multipliers

- IO Blocks

Slices are the most basic type of logic cell and are the foundation of FPGA technology. The flip-flops in a slice are used to implement register storage and the look-up tables are mainly used to implement logic, such as multipliers, adders or shifters. On recent generations of FPGAs the small look up tables can also be used to construct small on-chip RAMs called distributed RAMs. These distributed RAMs are an efficient storage mechanism for small arrays of data in an FPGA based design.

Recent FPGAs, such as the Xilinx Virtex-II used in this thesis, include two other types of logic cell; 18×18 bit hardware multipliers, and dual-ported RAMs, called block RAMs. Integer multipliers consume a lot of FPGA resources when constructed using conventional slices. The hardware multipliers provide a more efficient way to multiply integers on-chip. The block RAM cells are small 18 kilobit RAMs that are available for on-chip storage of data, and they substantially improve the data storage capacity of modern FPGAs. Sophisticated Input/Output blocks are the final type of logic cells. These cells connect the pins of the FPGA chip to the FPGA logic, and They support many different IO standards allowing FPGAs to communicate with many different devices.

FPGAs allow system designers to get the benefits of customising their processor architecture to the needs of their application, without the significant cost and risk of producing their own custom application specific integrated circuit (ASIC) chip. The non-recoverable engineering costs associated with a designing and producing a new ASIC chip grow with each new generation of silicon technology. The non-recoverable engineering cost for the latest 65nm silicon fabrication technology is now in the order of millions of dollars. For a chip using this technology to be commercially successful the manufacturer must sell a very significant number. So for small volume production custom ASIC chips are getting less and less attractive. Meanwhile these improvements in process technologies are making FPGAs larger, faster, cheaper and more capable than ever. Consequently FPGAs are becoming more and more attractive for a wide variety of fields where a custom ASIC processor would be technically beneficial but not financially viable. One such area is high performance computing.

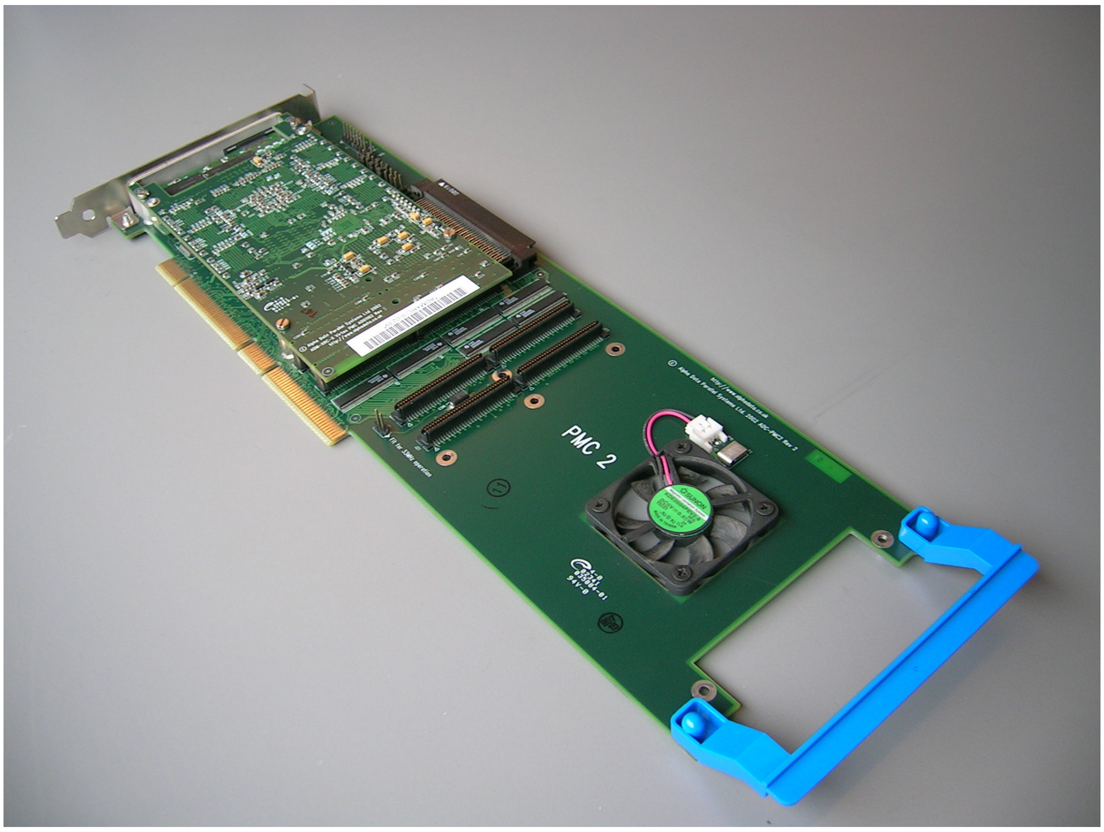


Figure 2-b. The ADC-PMC carrier card used in this work, with a single ADM-XRC-II development card fitted.

FPGAs give system designers access to significant levels of fine-grain instruction level parallelism and this parallelism has long been used to improve performance for application areas such as digital signal processing (DSP). FPGAs have been widely used to accelerate integer and fixed-point domain applications for some time. However, due to the significant device resources required by floating point arithmetic units, FPGAs have previously been too small to use for floating point calculations. FPGAs are now available that are large enough to support a significant quantity of floating-point arithmetic units on a single-chip. This raises the possibility of exploiting the parallelism of FPGAs for high performance computing applications.

2.1.1 The Alpha Data ADM-XRC II Prototyping Board

The designs in this work are implemented in hardware using FPGA prototyping boards supplied by Alpha-Data [Alpha-Data '05]. The boards used are Alpha-Data

ADM-XRC II boards. The ADM-XRC II board is also re-sold by Celoxica as the RC2000 prototyping board [Celoxica '06]. The ADM-XRC II board provides a large FPGA along with up to eight banks of off-chip SRAM memory for storing application data. The boards used in this thesis have six banks of SRAM memory each, providing a total of 12 or 24 Mega-Bytes of memory.

The boards themselves fit on to a ADC-PMC carrier card, which is a 64-bit PCI card that fits into a standard PCI slot in a PC computer. Figure 2-b shows an ADC-PMC card with one ADM-XRC-II card fitted to it. The ADC-PMC card can hold two ADM-XRC-II cards; the spare location can be seen at the front end of the card shown in Figure 2-b marked with “PMC 2”. A second ADM-XRC-II card was fitted to this location for the dual FPGA part of this work. The dual FPGA implementation is detailed in Chapter 7. The carrier card can accept up to two ADM-XRC II boards, each with their own FPGA and memory banks. Both boards connect to the host computer’s PCI bus through the carrier card. The ADM-XRC II boards used for this work have 6 banks of SRAM memory, and have a six million gate equivalent Xilinx Virtex II FPGA, a XC2V6000FF1152.

The carrier card provides an interconnection between two boards mounted on it. This interconnect was used to build the dual FPGA implementation of the FPGA based Dirac operator. The interconnection consists of 64 wires which can be used by a number of different communications protocols. The choice of communications protocol, which is made by the application designer, depends on the requirements of the application.

2.1.2 ADM-XRC-II Board Support Package

A package consisting of a software application programming interface (API) and a set of hardware macros is supplied with the ADM-XRC-II prototyping board. This gives the designer the ability to do the following:

- Transfer data between the host computer and the SRAM banks.
- Access the SRAM banks from within an FPGA design.

- Perform a blocking transfer of 32-bit data words between the FPGA and the host computer. This can be used to synchronise the host computer program with the FPGA design, since neither FPGA nor host can continue until the transfer is complete.

The API is used to write a software program - the host program - that runs on the host computer. The host computer programs the FPGA and manages data transfer to and from the FPGA and its SRAM banks. Hardware macros are included in the FPGA design that allow the host program to access the SRAM banks attached to the FPGA.

All FPGA to host communication is performed over the host computer's PCI bus, to which the ADM-XRC II FPGA board is connected through the carrier card. The ADM-XRC II board includes a PLXPCI chip which handles the complex logic required to interface to a PCI bus. The PLXPCI chip then interfaces with the FPGA design. This eliminates any need for an on-FPGA PCI controller, saving significant FPGA resources. The FPGA to PLXPCI interface logic is quite simple, and is included in the board support package. The Handel-C board support package ensures that the SRAM banks are clocked correctly and handles all data transfer between the host computer and the FPGA and its software.

The SRAM banks cannot be accessed simultaneously by both the host computer and the FPGA design. The status register must be used to synchronise the host and FPGA so that only one accesses the SRAMs at any given time. The following is an outline of the steps taken by a host program running an FPGA design on an ADM-XRC II board:

- Host program sets clocks and loads FPGA design.
- FPGA design performs a blocking status register read, stalling until the host writes to the status register. The FPGA does not access the SRAMs until after it has performed a successful read of the status register.
- Host transfers data to SRAM banks and then writes to the status register.

- FPGA reads the status registers and then begins calculation.

- Meanwhile, host performs a blocking status register read stalling the host until the FPGA writes to the status register

- Once the calculation is completed the FPGA writes to status register, the host is now free to read the result from SRAM

This basic scheme prevents simultaneous access to the SRAMs and is the scheme used for the designs in my work. My designs require little FPGA to host communication once the calculation is running making the simple host program sufficient for my needs.

2.2 Hardware design languages

In recent years, a number of higher level hardware description languages have been developed as an alternative to the traditional languages of Verilog and VHDL. Each type of tool has a domain of design types to which it is suited. High level languages are better suited to algorithmic hardware implementations whereas conventional HDLs are better suited to lower level more structural designs. This gives high level tools a number of advantages:

- *Support for Rapid Prototyping*. The time required to generate a working prototype design in a higher level language is significantly less than for conventional tools.

- *Higher Level of Abstraction*. Conventional HDLs such as VHDL are very detailed which is very useful for structural designs. Algorithmic implementations are more difficult in these tools as the design rapidly becomes unwieldy and complex. Higher level hardware languages help designers manage complexity in such designs.

These benefits lead to some disadvantages however. The main disadvantage is the relative immaturity of the tools; higher level hardware design languages are quite new and the quality of their development environments is far behind the development tools for VHDL or Verilog. A good example is the difficulties designers face when trying to use third party Intellectual Property (IP) with Handel-C designs. It is very common for FPGA designs to include a number of precompiled pieces of logic called hardware macros. These hardware macros give designers access to a large library of highly optimised designs.

A very large number of "black box" hardware macros with associated simulation files are available for VHDL/Verilog design tools, however whilst the hardware macros can be easily integrated into a Handel-C design, the Handel-C simulator cannot use the VHDL/Verilog simulation files. It is thus very difficult to simulate Handel-C designs that make use of external hardware macros. The DK environment provides a proprietary simulation mechanism for simulating external hardware macros, but since Handel-C is new, very few hardware macros supply accompanying Handel-C simulator files.

2.2.1 Handel-C

Handel-C is a hardware development language based on the concept of communicating sequential processes (CSP) [Hoare '85]. CSP allows the description of systems that consist of multiple sequential components which are executed in parallel. Communication and synchronisation between the processes are provided by a synchronous message passing system.

Occam is a programming language that is based around the CSP model [INMOS. '84]. Occam provides channels that are used to communicate between parallel processes. Handel-C is a development of the Occam language and its structure is derived from that of Occam. Handel-C provides synchronous communication channels and the ability to specify that code segments be performed sequentially or in parallel.

The syntax of Handel-C is heavily derived from that of the C language. C syntax was chosen as a base for Handel-C syntax since it is a popular and familiar language. The syntax is adapted with some additions that allow it to address the specific architecture of FPGAs. The most significant additions are the ability to specify parallelism, the ability to specify the size in bits of all variables used in the design and the provision of synchronous channels for communicating between parallel parts of the design. Handel-C can be compiled for the cycle accurate Handel-C simulator or for hardware by compiling it into an EDIF net-list, which is placed and routed for an FPGA using the FPGA manufacturer's place and route tools. The Handel-C compiler can also convert Handel-C source code to VHDL or Verilog source code.

2.2.2 Basic use of Handel-C

```
unsigned int 13 a, b, c, d, e, f, z;

par{
    a=a*b;
    y=c*d;
    z=e*f;
}
```

Figure 2-c. Use of Handel-C par{} block to specify parallelism

This section introduces the most important aspects of Handel-C and demonstrates how they are used for FPGA design. The designer must the width in bits of all variables in Handel-C at compile time. This allows the designer to save significant FPGA resources by keeping variables as small as possible. For example Boolean values can be stored using one bit variables, instead of the full 32 bits that are used in software. Distinct FPGA hardware is created for each variable declared in a Handel-C design; they are not automatically reused like a processor register. The code in Figure 2-c shows the declaration of several thirteen bit wide unsigned integer variables.

```
unsigned 64 a[1000], b[1000], c[1000];
unsigned 16 beta;
unisgned 8 x;

for(x=0; x<1000; x++){
  a[x] = a[x] + (b[x] * (c[x] * (unsigned 32)(0 @ beta)));
}
```

Figure 2-d. A complex expression causing a low clock rate

Every assignment in Handel-C requires one clock cycle to execute. The exception is when statements are placed within a *par* block. The Handel-C *par* construct is the principal tool for exploiting parallelism in a Handel-C design. Statements grouped within a par block are performed in parallel with each statement using separate hardware. Figure 2-c shows three multiplications performed in parallel in a single clock cycle using a *par* block. In essence, a *par* block specifies a fork-join model of parallel computation.

If the par block were removed from Figure 2-c the multiplications would take three cycles to complete. Removing the par block would not reduce resource requirements however since separate hardware would still be used for each of the three multiplications. A function or a shared expression could be used to perform the three multiplications using the same hardware on three subsequent cycles. This would lower the resource requirements of the design.

```
unsigned 64 a[1000], b[1000], c[1000];
unsigned 64 ct, bt;
unsigned 16 beta;
unisgned 8 x;

ct = c[0] * (unsigned 64)(0 @ beta);

par{
  bt = b[0] * ct;
  ct = c[0] * (unsigned 64)(0 @ beta);
}

x=0;
while(x<1000){
  par{
    a[x] = a[x] + bt;
    bt = b[x+1] * ct;
    ct = c[x+2] * (unsigned 64)(0 @ beta);
    x++;
  }
}
```

Figure 2-e. Pipelined and parallelized Handel-C loop

Figure 2-d shows a code fragment where every item in an array must be updated. However the update logic is complex, and causing a slow clock rate. The poor clock rate can be improved by splitting the update into several steps. These steps can be parallelised using pipelining, as shown in Figure 2-e. By splitting the single complex assignment into three stages the clock rate of the design has improved significantly, and since the three stages are pipelined, the number of cycles required to update the array has not increased significantly. The code in Figure 2-e has also significantly improved performance by updating the loop counter in parallel with the loop body, whereas in Figure 2-d it is updated sequentially.

2.2.3 Data Storage in Handel-C

Most FPGA designs cannot store all their data in register type storage; usually some form of array type storage is required. FPGAs provide three types of array storage; arrays of registers, distributed RAM and block RAM.

Register arrays

Register arrays are constructed from sets of registers and are the most flexible type of array storage, since there are no parallel access restrictions except that only one write can be made to each element in a single clock cycle. However they also consume far more resources than either of the RAM storage types, due to the multiplexers required to control access to the array. Arrays of registers should not be used except when absolutely necessary, or when the arrays involved are very small (around four to six elements).

Distributed RAM

Distributed RAM is constructed only from LUTs and uses no flip flops at all (see section 2.1 for an explanation of FPGA technology). Distributed RAM uses substantially less FPGA resources compared with register arrays, but at the cost of a restricted number of possible parallel accesses. Distributed RAM can be either single or dual ported. Single ported RAM only allows one access (either a read or a write) to the entire RAM per clock cycle. Dual ported RAM is created using two identical parallel distributed RAMs, so it is a less efficient storage mechanism than single ported RAM. Dual ported distributed RAM allows two reads or one read and a write per cycle. It is not possible perform two writes to a dual ported distributed RAM in a single clock cycle.

Block RAM

Block RAM is constructed using the on-chip 18-kilobit block RAMs that are part of recent Xilinx FPGAs, starting with the Virtex II family. Block RAM is true dual ported RAM, in that it allows two operations of any type, including two writes, per cycle. The dual porting of block RAM has no extra cost, as is the case for dual ported distributed RAM.

Access to block RAM is through a one cycle pipeline. To read data from a block RAM the data address is presented to the block RAM, which then writes the data into output registers attached to the block RAM ports. Thus the data is available for use in the next clock period. This one cycle pipeline makes block RAM access more complicated than accessing distributed RAM, which has single cycle access characteristics. To hide this complexity Handel-C clocks block RAMs using a double speed clock, which makes the block RAMs seem to have single cycle access characteristics. Unfortunately this tends to severely limit the clock rate of Handel-C designs that use block RAMs. Later versions of Handel-C include a compiler optimisation that converts block RAMs to pipelined access if data is only read from a block RAM into a single register that is not written to from anywhere else in the design. This optimisation improves clock rate significantly for Handel-C designs that use block RAM.

2.3 Number Representations

IEEE floating point is the standard approach for performing non-integer calculations and is implemented on most commodity processors. Log arithmetic, also frequently referred to as the Logarithmic Number System (LNS), is an alternative approach to these calculations that uses fixed point logarithms to represent non-integer numbers. Log arithmetic requires vastly fewer resources for multiplication and division compared with IEEE floating point but addition and subtraction become significantly more complicated. When this project began there were no available options for performing conventional floating point on FPGAs, however log arithmetic was available for use on the project. Consequently it was decided to investigate this system.

2.3.1 Single Precision Log and Floating Point

The logarithmic arithmetic designs use commercial LNS cores from the High-Speed Logarithmic Arithmetic system (HSLA) by Matousek et al [Matousek '02]. These are 32-bit logarithmic cores, which support the full range of exceptions from the IEEE

	LNS Adder	Underwood Adder
Slices	1648	496
Multipliers	8	0
Block RAM	28	0
Latency (Clock cycles)	8	13
Clock Rate (MHz)	90	165
Pipes per FPGA (Xilinx Virtex-II-6000)	10	68

Table 2-a. Comparison of the resource requirements of *two* log arithmetic adder pipelines and a comparable IEEE floating point adder pipeline

floating point standard. The cores are highly-optimized for both performance and space.

Table 2-a, Table 2-b and Table 2-c show the resource requirements of the LNS adder, multiplier and divider respectively compared with the requirements for comparable IEEE arithmetic units published by Underwood [Underwood '04a]. The pipeline latencies of the Underwood cores are variable. The cores are designed so that the pipeline latency can be reduced however this also reduces the maximum clock rate the cores can run at. Short pipeline latencies are desirable for designs where there are a lot of dependencies between calculations. Using arithmetic cores with long pipeline latencies for such designs can hurt overall performance since the arithmetic pipelines will spend a lot of time waiting for the results of previous calculations.

Logarithmic arithmetic has a clear advantage for multiplication and division. The LNS multiplier is substantially smaller than Underwood's multiplier and has a lower latency. In addition the fully pipelined LNS divider returns similar performance to Underwood's divider but uses less than 5% of the resources and has a much lower

	LNS Multiplier	Underwood Multiplier
Slices	83	598
Multipliers	0	4
Block RAM	0	0
Latency (Clock cycles)	1	16
Clock Rate (MHz)	250	124
Pipes per FPGA (Xilinx Virtex-II-6000)	407	36

Table 2-b. Comparison of the log arithmetic multiplier and a comparable IEEE floating point multiplier

latency.

The penalty for the small and low latency LNS multiplier and divider is that a pair of log adders requires 66% more slices than a pair of IEEE adders, and a significant quantity of block RAM and hardware multipliers. This block RAM requirement means a maximum of 10 log adder pipelines can fit on the Xilinx Virtex-II-6000 FPGA used in this thesis. However ten log adder pipelines use only 25% of slices on the Xilinx Virtex-II-6000 FPGA, leaving plenty of space for control logic and other arithmetic units.

The resource requirements for the LNS cores favour applications that have a high proportion of multiplications compared with additions. Also the very small size of the divider is a particular advantage for applications which use division only rarely. The lattice QCD conjugate gradient solver implemented in this thesis is an example of such an application. A large IEEE divider is a waste of resources in such designs. Section 3.4 gives an overview of the history and development of logarithmic arithmetic, and compares the current state of the art for floating point and log arithmetic cores.

	LNS Divider	Underwood Divider
Slices	82	1929
Multipliers	0	0
Block RAM	0	0
Latency (Clock cycles)	1	37
Clock Rate (MHz)	250	100
Pipes per FPGA (Xilinx Virtex-II-6000)	412	17

Table 2-c. Comparison of the resource requirements of the log arithmetic divider and a comparable IEEE floating point divider

The Matousek cores were chosen to implement the logarithmic arithmetic FPGA implementations of the lattice QCD Dirac operator and the logarithmic arithmetic conjugate gradient application that are described in Chapter 5. The above comparison of the Matousek cores with floating point arithmetic cores and the comparison between the Matousek cores and alternative logarithmic arithmetic cores in section 3.4.7 show that they are competitive with alternative systems. When the work described in this thesis started the Matousek cores were the only non-integer arithmetic cores available to the project, consequently they were chosen as the best option. During the project single precision floating point arithmetic cores became available however work on the logarithmic arithmetic cores was well advanced by this time. Upon completion of the log arithmetic designs described in Chapter 5, the Moloney double precision floating point cores became available [Moloney '04]. Since double precision arithmetic is the standard arithmetic system for high performance computing applications, these cores were chosen for further work, instead of investigating single precision floating point arithmetic cores.

2.3.2 Double Precision IEEE Floating Point

The IEEE double precision implementations in this work use Moloney’s cores

[Moloney '04]. The cores are IEEE compliant, fully pipelined and highly optimized. They achieve good clock rates with short pipeline latencies, which is particularly important for applications with long chains of data dependencies. In such applications arithmetic units must often wait for operands which are the output of previous operations. This causes the arithmetic units to lie idle for a significant portion of the time. If the arithmetic pipeline latencies are long then the waiting time is increased, causing a loss in performance.

Table 2-d, Table 2-e and Table 2-f show performance comparisons for the double precision adder, multiplier and divider compared with cores published by Underwood [Underwood '04a]. In each case, the Moloney units compare well with the Underwood units. The Moloney adder has a slower clock rate than the Underwood adder, however its pipeline is significantly shorter and it uses fewer resources. Similarly the Moloney multiplier has a shorter pipeline latency and a slower clock rate than the Underwood cores. The Moloney core also uses significantly fewer FGA resources.

The Underwood cores implement de-normalised number handling which requires significant quantities of FPGA resources. I use a version of the Moloney cores that do not handle de-normalised numbers which reduces resource requirements compared with other versions that do handle de-normalised numbers. De-normalised number handling is not needed for lattice QCD calculations, since the range of the numbers used is limited. De-normalised number handling is important in applications where very small numbers are involved; without de-normalised number handling, any results smaller than the smallest represent-able normalised value will underflow, that is they will be set to zero. De-normalised numbers fill in the “gap” between the smallest normalised number and zero and reduce the incidence of underflow where calculations are performed involving numbers with very small magnitudes.

The two divider cores are very different in their implementations. The Moloney core is designed for systems where division is used very infrequently, so it was designed to be small and to be able to perform a single divide within a small number of cycles. However it is not pipelined so it can only perform one divide at a time. In contrast

	Moloney	Underwood
Slices	937	1090
MHz	110 (Speed grade 6)	125 (Speed Grade 5)
Latency	6	14

Table 2-d. Comparison of double precision adders

	Moloney	Underwood
Slices	825	1607
Mult 18x18	9	9
MHz	114 (Speed grade 6)	105 (Speed grade 5)
Latency	7	20

Table 2-e. Comparison of double precision multipliers

	Moloney	Underwood
Slices	1789	6858
MHz	95 (Speed grade 6)	83 (Speed grade 5)
Latency	36 (Not Piped)	67 (Pipelined)

Table 2-f. Comparison of selected dividers

the Underwood divider is designed for systems where there is a high demand for divide operations. The Underwood divider is fully pipelined, which is unusual for a floating point divider. It has a long pipeline latency so it takes twice as long to perform a single divide operation, but being fully pipelined it has a throughput over 30 times greater than the Moloney core. The cost of this performance is that it is nearly 4 times larger than the Moloney divider.

The Moloney divider was used to implement a double precision version of the lattice QCD conjugate gradient solver. Divides constitute a tiny fraction of the floating point operations in this application. Two divides are issued per iteration of the main application loop, and several thousand cycles separate each issue. Thus a small divider with relatively poor performance is far better suited to the needs of the application that a large divider with unnecessarily high performance. Consequently the Moloney divider is far better suited to the needs of the conjugate gradient applications.

2.4 Summary

This chapter has introduced the most important technologies that have been used in this work, including FPGAs, FPGA development boards, high level hardware design languages and the non-integer arithmetic FPGA cores used in my designs. The next chapter introduces lattice QCD, the application that is studied in this work, together with an extensive survey of the literature relating to lattice QCD computing machinery and to non-integer arithmetic on FPGAs.

Chapter 3

Literature Survey

Lattice QCD is a significant high profile high performance computing application, making it an ideal example application for evaluating a platform's suitability for high performance computing. Creating computing machinery for lattice QCD is the focus of significant world-wide research effort. By comparing my results with this body of research, I am able to draw comparisons between FPGAs and the state of the art for high performance computing.

This chapter begins with a description of quantum chromodynamics (QCD) theory and an explanation of why computer simulations must be used to help validate this theory. This use of computer simulations is called lattice QCD. The chapter then continues with a survey of the current state of the art for lattice QCD computing machinery. This survey examines three categories of machine; custom ASIC based supercomputers, commercial supercomputers and PC clusters.

The use of FPGAs for high performance computing is only possible due to the recent availability of FPGA based non-integer arithmetic cores. Two non-integer arithmetic systems are used in this project. They are single precision equivalent logarithmic arithmetic and IEEE double precision floating point. The development of both of these arithmetic systems is described in this chapter, including the recent development of FPGA based implementations of the two systems. Finally, recent research into the use of FPGAs for high performance computing is described in detail.

3.1 QCD and Lattice QCD

3.1.1 Quantum Chromodynamics

All of the matter in the universe is made of atoms. There are many different types of atoms and all are constructed from some combination of electrons, protons and neutrons. However these subatomic particles are themselves constructed from other particles that are even smaller, called quarks. There are several different types of quark which are combined together to form larger particles called hadrons. Protons and neutrons are both examples of hadrons. The quarks themselves are not stationary within a hadron however. They move at nearly light speed making up a "hazy buzzing cloud of action" [Davies '98]. An important question is what holds these quarks together within a hadron. The answer lies with something called "The Strong Force of Nature".

Physicists already have an excellent understanding of electrodynamic force. This force is described by quantum electrodynamic theory or QED for short. QED describes how electrodynamic force is caused by two particles, for example two electrons, exchanging a photon. This exchange photons causes attraction or repulsion between magnetic objects. In a similar fashion to QED theory, quarks interact by exchanging gluons and it is this exchange of gluons that causes the force which binds quarks together within a hadron.

Quantum Chromodynamics or QCD is the quantum theory that describes how this exchange of gluons occurs. It is based on QED which is well established and has been shown to be very accurate. In QCD both the quarks and the gluons carry a charge; in comparison, photons in QED have no charge. Also charge in QCD comes in one of three "colours"; red, green or blue. QED has only one type of charge. It is the presence of the three colours of charge that gives quantum chromodynamics its name. Each of the three colours of charge can be either the colour charge or the anti-colour charge, for example a particle may have a "red" charge which would be attracted to a particle with an "anti-red" charge. This is similar to charge in QED where the single types of charge can be positive or negative.

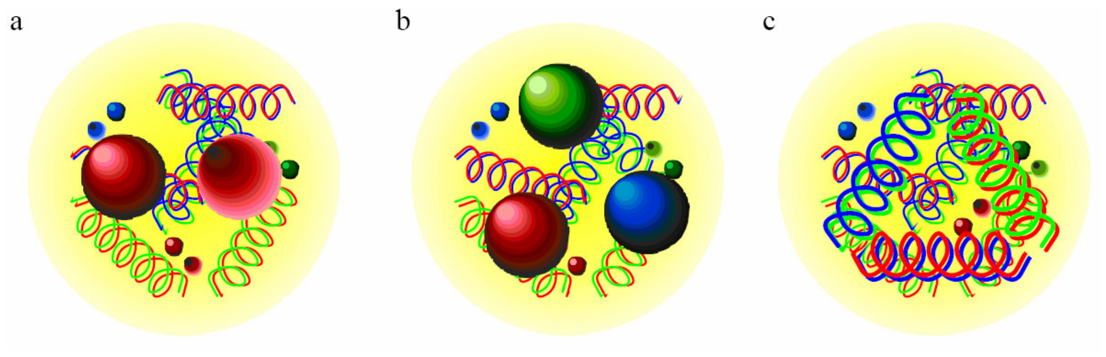


Figure 3-a. The three types of hadrons that can exist in nature; Mesons, Baryons and Glueballs [Davies '00].

Proving the accuracy of QCD is very difficult however; not least because quarks cannot exist in an unbound state, they can only exist when bound in colourless hadrons. This is a phenomenon called confinement which is just one of the things that QCD is believed to explain.

Figure 3-a shows the three types of hadrons that can exist in nature.

- Mesons (particle a) are made of a quark with a particular colour charge and an anti-quark with the opposite anti-charge.
- Baryons (particle b) are constructed from three quarks, each with different colour to the others.
- Glueballs (particle c) are made from a collection of gluons, which have no overall net charge.

In QCD the strength of the force between quarks grows as the quarks move further apart. In comparison in QED theory the electrodynamic force between the particles becomes weaker when you pull two electrons away from each other. This is because it is more difficult for the electrons to exchange photons thus weakening the force between the two particles. If the distance between them is large enough then their electric fields become completely separate and no force at all exists between the two particles.

In QCD it is not only the quarks that can exchange gluons; the gluons themselves also carry a charge meaning that a pair of gluons can also exchange gluons.

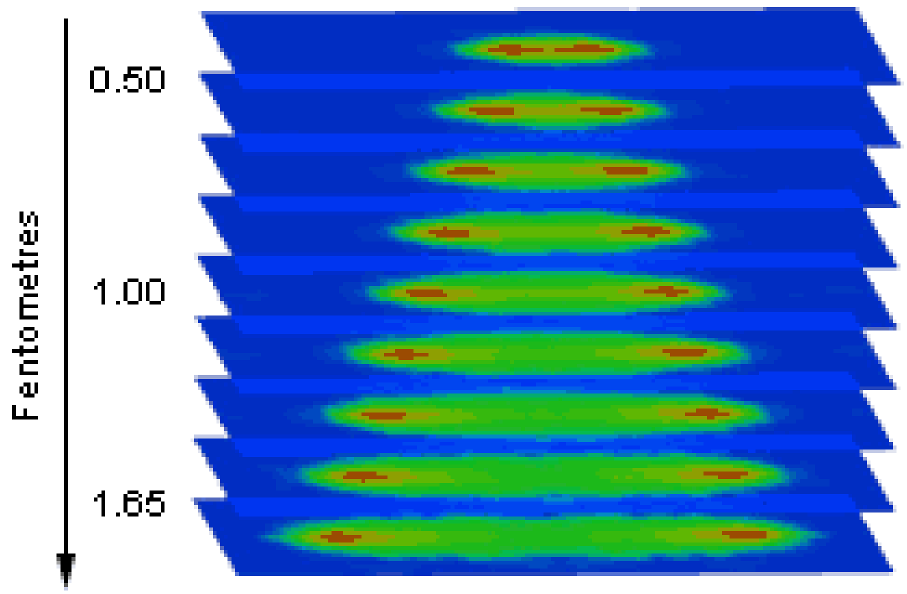


Figure 3-b. A lattice calculation showing how the strength of force develops as you pull a quark and an anti-quark apart, which prevents them escaping from one another [Davies '03].

Furthermore these exchanged gluons can exchange still more gluons and so on. As the quarks are pulled further apart the number of gluon exchanges increases making the strength of the force increase with distance, confining the quarks within hadrons. This phenomenon is known as confinement. Figure 3-b shows graphically how the force between two QCD particles grows as the two particles are pulled apart.

3.1.2 Lattice QCD

Confinement means that no free quarks or gluons have ever been observed experimentally. As a result numerical simulations of QCD performed on high-performance computers have been used for over thirty years in an attempt to make *ab initio* predictions about experimental results using QCD theory. The predictions can then be compared with the hadrons observed from experiment in order to test QCD theory. This is done because the mathematical complexity of QCD, caused by the ability of the gluons to exchange charge, makes it impossible to solve using traditional methods. Consequently the only way physicists can test QCD theory is to use massive computer simulations to compare the theory with experimental results.

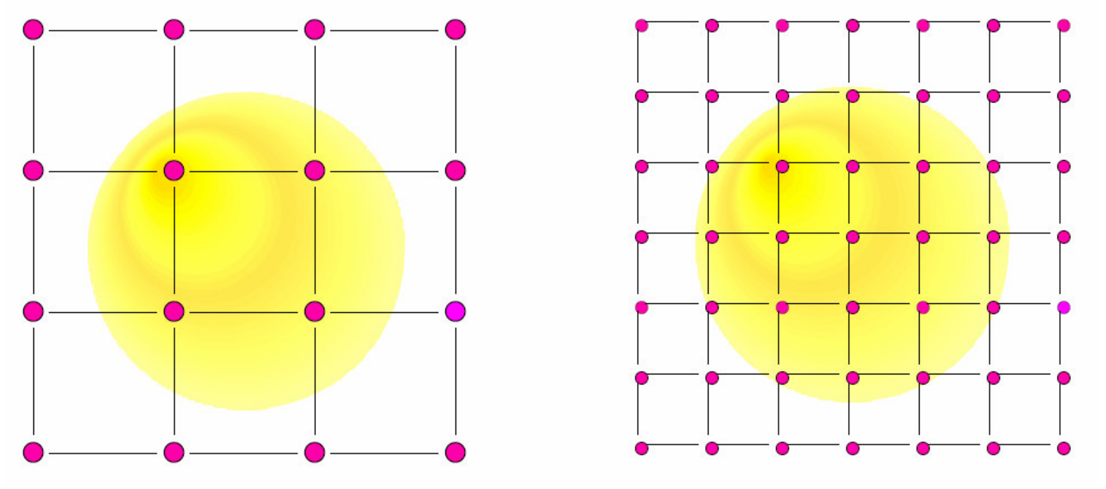


Figure 3-c. Diagram showing how a lattice is used to simulate space and time around a hadron. The lattice must be larger than the hadron in all dimensions. A lattice with more points (the lattice on the right) will give a more accurate result but will take more computer time to simulate.

The application of computer simulations to QCD in this way is called lattice QCD.

Lattice QCD simulations provide vital inputs into experimental searches for new physics at ever-increasing energy scales in the world's largest particle collider experiments. It helps physicists predict how and where new types of particles are likely to be found. The simulations also attempt to explain the mechanism for confinement in QCD and so bring us to a better understanding of the fundamental nature of matter itself.

In practise lattice QCD simulates a small continuous region of space-time using a discrete grid or lattice. This lattice represents a very small portion of space and time that is about twice the width of a proton. Figure 3-c shows a graphical representation of two dimensions of two lattices. The left hand lattice has fewer points than the right hand lattice and so will be faster to solve, but will give a less accurate result.

The quark gluon interactions are simulated by placing the quarks on the sites in the lattice and the gluons on the links between these sites. A lattice QCD simulation randomly places a few quarks and gluons on the lattice and then uses the rules of QCD to simulate their behaviour. These simulations are repeated many times so that the results show what the average behaviour of the quarks is. The complexity of the quark to quark interactions means that these simulations are extremely computationally intensive.

Indeed current lattice QCD simulations mostly use the quenched approximation where only quark to quark interactions are simulated. This reduces the computational complexity of the simulations to a more manageable size. Unfortunately it is impossible to investigate the internal structure of light hadrons, such as protons, using the quenched approximation. Gluon to gluon interactions are far more important for these lighter hadrons so physicists need a new generation of machines that can supply enough processing power to perform simulations that include these interactions.

Lattice QCD belongs to a general class of high performance computing algorithms called sparse matrix solvers. Other examples of sparse matrix solvers include Computational Fluid Dynamics, Finite Element Solvers and certain astrophysics applications. However lattice QCD is crucially different to these applications because in lattice QCD the matrix is constant whereas it must be re-formulated for every calculation in other applications. This allows the matrix representation to be built into the algorithm itself and so it is not explicitly represented. This allows lattice QCD calculations to sustain much higher performance compared with other sparse matrix solvers. Also the complexity of the quark to quark interactions dramatically increases the computational requirements of lattice QCD compared with other sparse matrix solvers.

The lattice for a lattice QCD calculation is represented by a set of large matrices. These matrices are hyper-cubes of four dimensions. The number of elements in these matrices is determined by (1).

$$NS = NX \times NY \times NZ \times NT \tag{1}$$

The four values, NX, NY, NZ and NT, dictate the number of points to be simulated in each of the three dimensions of space and also the dimension of time. These four quantities determine the number of points in a lattice QCD application. Current lattice QCD simulations simulate over two million points in space-time. Obtaining a single scientific result for such a simulation using the quenched approximation needs approximately 6.6×10^{15} floating-point operations [Gellrich '03]. As such the computational requirements for lattice QCD are very significant and will continue to grow in the future. Physicists need to stop using the quenched approximation in order to get more accurate results and will need ever more powerful machinery for this task.

3.1.3 Lattice QCD Dirac operator

The Dirac operator is the most significant part of a lattice QCD calculation. Mathematically it multiplies a sparse matrix by a vector; however the actual code for the algorithm has little in common with conventional matrix-times-vector multiplication codes. The matrix in lattice QCD is constant for all calculations so, to improve performance, the matrix representation is built into the algorithm itself. Consequently memory access in the Dirac operator is completely predictable, which delivers a substantial performance improvement over conventional sparse matrix-times-vector multiplication. In comparison other sparse matrix solvers, such as Finite Element Solvers, must be able take any matrix as an input. Consequently their memory access patterns are unpredictable, resulting in performance that is limited by memory bandwidth and cache behaviour.

The complexity of the quark to quark interactions that lattice QCD models means that both the dataset for the application and the calculation itself are much more complicated than for other sparse matrix solvers. The vectors for lattice QCD are represented using arrays of small matrices of complex numbers; consequently the computational complexity is much higher than for other sparse matrix solvers where the vectors consist of simple arrays of numbers.

3.1.4 The conjugate gradient method

Conjugate gradient solvers are one of the main types of application used in lattice QCD. Conjugate gradient methods are iterative solvers which start with an approximate solution to a system of equations, which is then refined over multiple iterations until it is equal, or at least sufficiently close, to the true solution. Conjugate gradient methods provide a general means of solving systems of linear equations, such as that shown in (2), where A is the system of equations expressed in matrix form, y is a known vector and x is the unknown solution to the system.

$$A \cdot x = y \tag{2}$$

$$f(x) = \frac{1}{2} x \cdot A \cdot x - y \cdot x \tag{3}$$

$$\nabla f = A \cdot x - y \tag{4}$$

The idea of conjugate gradient is to minimise the function (3). This function is minimised when its gradient, calculated using (4), is zero. In order to minimise (3), a succession of search directions $p^{(i)}$, and improved minimisers $x^{(i)}$, are generated. This process will iteratively find the solution to (2) [Press '92].

$$\alpha = \alpha_i = r^{(i-1)^T} r^{(i-1)} \big/ p^{(i)} A p^{(i)} \tag{5}$$

$$x^{(i)} = x^{(i-1)} + \alpha p^{(i)} \tag{6}$$

$$r^{(i)} = r^{(i-1)} - \alpha q^{(i)} \text{ where } q^{(i)} = A p^{(i)} \tag{7}$$

$$\beta_i = r^{(i)^T} r^{(i)} \big/ r^{(i-1)^T} r^{(i-1)} \tag{8}$$

$$p^{(i)} = r^{(i)} + \beta_{i-1} p^{(i-1)} \tag{9}$$

The conjugate gradient method generates successive approximate solutions, $x^{(i)}$, using (6). Also generated are residuals that correspond to these approximations, $r^{(i)}$, suing (7) and search directions, $p^{(i)}$, using (9) that are used to update both the approximate solutions and the residuals. Two update scalars, α_i and β_i, are computed, using (5) (8) respectively, on every iteration of the method; these scalars are used to make the successive vectors obey certain orthogonality conditions. The sequence of vectors generated by the method is a conjugate (or orthogonal) sequence; this is what gives the conjugate gradient method its name [Barrett '94].

3.1.5 Conjugate gradient in lattice QCD

The conjugate gradient method is used extensively in lattice QCD. The source code for the main loop of the conjugate gradient solver that is the basis of the conjugate gradient designs described in this thesis is shown in Figure 3-d. Comments in the code show which mathematical equation the segments of code in the main loop are equivalent to. The names of the operators used have been renamed for clarity. The original names are used in the full source code, which can be found in Appendix C.

```
t_gl3 g[NS][4];
t_wfv x[NS], y[NS], r[NS], p[NS], tmp1[NS], tmp2[NS];

kappa = 0.124;

GenGl3Vector(g, 1);
GenZeroWfvVector(x);
GenZeroWfvVector(y);
y[0][0][0].r = 1.0;

CopyWfvVector(r,y);
CopyWfvVector(p,y);

res_old = LatDotWfv(r,r);

while (res_old > 1.0e-6)
{

  //Calculate alpha - Equation (5)
  Dirac(tmp1, kappa, g,   p);
  alpha = res_old / DotProduct(tmp1, tmp1);

  //Update x - Equation (6)
  AddScaledVector(x,  alpha,    p);

  //Update r - Equation (7)
  Dirac (tmp2, kappa, g,tmp1);
  AddScaledVector (r, -alpha, tmp2);

  //Calculate beta - Equation (8)
  res_new = DotProduct(r,r);
  beta = res_new / res_old;

  //Update p - Equation (9)
  ScaleAndAddVector(p, beta, r);

  res_old = res_new;
}
```

Figure 3-d. Main loop of lattice QCD conjugate gradient solver

Thus calculating *alpha* requires a call of the Dirac operator (which is a matrix-times-vector multiplication) and a call of the dot product operator on the result. This is equivalent to equation (5) above. Updating *x*, equivalent to (6), involves scaling the vector *p* by *alpha* and adding it to *x*; this is done using the *AddScaledVector* operator.

Updating *r*, which is equivalent to (7), is done through a call to the Dirac operator and the result is scaled by minus *alpha* and added to the existing value for the *r* vector. The new value for *beta* is calculated by dividing the result of a dot-product operation on *r* by the previous residual sum; this is equivalent to (8). Finally the new value for *p* is calculated by scaling the vector *p* by beta and then adding the vector *r* to it; this operation is equivalent to (9).

3.1.6 Operations and data for the lattice QCD conjugate gradient solver

The dataset for conjugate gradient is made up of large matrices of small complex number matrices. There are two types of small matrix, *gl3* (3×3 elements) and *wfv* (4×3 elements). These are used to construct two different types of larger matrix. The first type is a matrix of *NS*×*4 gl3* matrices, which will be called the *g* type matrix, and the second is a matrix of *NS wfv* matrices, which will be called *y* matrices. The conjugate gradient dataset consists of one *g* matrix along with several *y* matrices.

The conjugate gradient solver uses three operations, which are:

- Dirac operator
- AddScaledVector and ScaleAndAddVector
- Dot Product

The Dirac operator is the most computationally intensive operation. The Dirac operator updates each point in a *y* type matrix taking another *y* type matrix and a *g* type matrix as operands. The dataset for a single call of the Dirac operator therefore consists of a *g* type matrix, two *y* type matrices and a single scaling value called *kappa.* It is constructed from 4 operations, which operate on the *gl3* (3×3) and *wfv* (4×3) matrices. They are:

- *Gamma* operators
- Matrix-multiply; *wfv* × *gl3*= *wfv*
- Matrix addition/subtraction; *wfv* + *wfv* = *wfv*

- Matrix scale; *wfv* × *value* = *wfv*

The *gamma* operators multiply a *wfv* matrix by a pre-defined constant matrix to produce a *wfv* matrix. The pre-defined matrices are constant for all runs of the application. The structure of the constant matrices is similar to an identity matrix, with only one number on each matrix row, and each number is either one or minus one. Consequently instead of implementing a full matrix multiply, the *gamma* function is instead implemented using a short series of additions and subtractions. This is why lattice QCD has higher performance than other sparse matrix solver based applications such as Computational Fluid Dynamics. In these other applications the matrix must be reformulated for every run making it impossible to build the multiplication into the algorithm in the manner of lattice QCD.

The Dirac operator uses 8 slightly different versions of *gamma* operators. The *wfv* × *gl3* complex number matrix multiply is the most compute intensive part of the calculation needing 264 floating point calculations to multiply a single pair of matrices. The matrix addition is a straightforward matrix addition. The matrix scale scales every element in a *wfv* matrix by a particular value. Internally each of these blocks has exploitable parallelism. The matrices are matrices of complex numbers, so each number has a real and imaginary component. The real and imaginary components of the numbers can be calculated in parallel. Also many of the blocks are independent and thus can be parallelized.

The *gamma* and matrix multiply blocks are paired and are referred to here as *gamma-mul* pairs. The eight pairs are independent and can be performed in parallel given sufficient resources. The *wfv* add blocks accumulate the eight results of the gamma-mul blocks into one *wfv* matrix. Parallelism is possible here by using a binary tree type reduction operation to perform the accumulation. A reduction operation can be completed in $\log_2(n)$ stages, where n is the number of components to be added together. There are 8 gamma-mul results, which can be added together in 3 (i.e. $\log_2(8)$) steps.

The Matrix Add-Scale and Matrix Scale-Add operations are very similar and are shown in equations (10) and (11). Both operators scale both the real and imaginary component of each number in a *y* type matrix and then add the result to the real and imaginary components of the same point in another *y* type matrix. The result of this is then stored in one of the two operand matrices depending on which operation is performed. The operations can be expressed by the following equations.

$$y = (y \times r) + x \tag{10}$$

$$y = y + (x \times r) \tag{11}$$

Dot product as used in this version of conjugate gradient separately squares the real and imaginary components of each number in a matrix of *wfv* matrices and accumulates the results onto a running total. This is done for the every iteration's result and the result is compared with that from the previous iteration. This gives a guide to how quickly the algorithm is converging; the greater the difference the greater the rate of convergence. Once the difference drops below a certain threshold the calculation is complete.

3.2 Lattice QCD Computing Machines

The massive and increasing computational demands of lattice QCD simulations have made the design and construction of computing machinery for lattice QCD an area of considerable research effort. The main priority for the designers of these machines is to maximise performance for the money available. Generally researchers have limited funds when they purchase a dedicated lattice QCD machine and they want to get the greatest performance for their money.

Commercial supercomputers give excellent performance for lattice QCD simulations, but they are usually expensive and are usually owned by large research institutions who share access between large numbers of research interests, limiting the amount of time available for lattice QCD simulations. However they are used for lattice QCD and some research groups have bought commercial supercomputers, such as IBM's BlueGene/L system, specifically for lattice QCD simulations. The performance of two types of commercial supercomputer, the SGI Altix and IBM's BlueGene/L are discussed by Wettig [Wettig '05].

As an alternative to commercial supercomputers two separate groups have developed systems which are both customised to the requirements of lattice QCD. Both of these projects aim to deliver performance that rivals commercial supercomputers, but at a much lower cost. By customising the systems, performance is improved and researchers can get more performance for their money. The most recent versions of these machines are apeNEXT [Belletti '06] and QCDOC [P. A. Boyle '05].

More recently PC processors have seen substantial improvements in their floating-point capabilities and these improvements have made PC type processors a viable platform for lattice QCD simulations. Significant research effort has been expended on investigating the use of PC hardware for lattice QCD, with excellent results [Holmgren '05b] [Gellrich '03]. A significant amount of this effort has been directed towards making use of the floating point vector processing capabilities of modern PC processors.

3.2.1 Commercial Supercomputers

Commercial supercomputers are generally bought by large research institutions for users from a wide variety of research areas; consequently they are optimised to give good performance for a wide variety of applications. Lattice QCD has been implemented on two of the latest commercial supercomputers, the SGI Altix and IBM’s BlueGene/L [Wettig '05].

The SGI Altix is based on the Intel Itanium 2 processor, which is a VLIW processor [Ellis '86]. The architecture of the system is shown in Figure 3-e and consists of a large number of compute nodes, each of which has two processors connected to memory, communications and IO sub-systems through a custom built super hub chip (SHUB). The nodes are connected together using SGI’s NUMAlink system, which is based on the ccNUMA (cache coherent Non-Uniform Memory Access) architecture, which allows shared memory domains of up to 512 CPUs. This allows SGI Altix systems consisting of thousands of processors to be built.

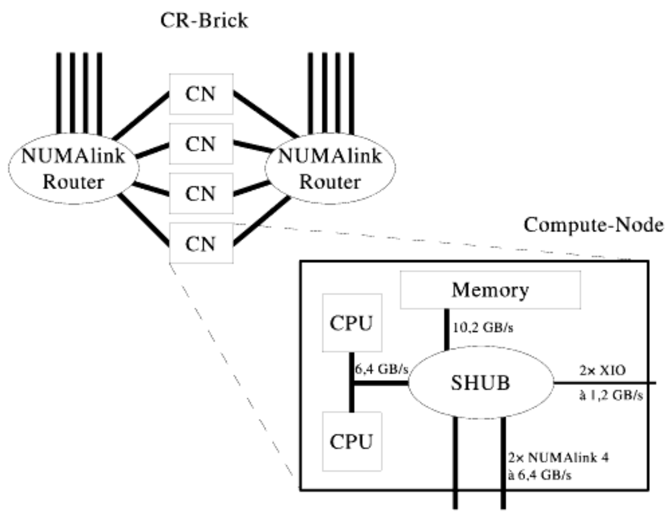


Figure 3-e. SGI Altix Architecture [Wettig '05]

Benchmark results quoted by Wettig show that for a small global problem size SGI Altix nodes can sustain 28% of peak performance for a system of 32 dual-core CPUs on the core Dirac operator, which translates to about 6 GFLOPs per dual-core CPU. This figure is expected to be halved for a larger number of nodes, giving a figure of 3 GFLOPS per dual-core CPU [Wettig '05]. These figures are impressive, however the per-processor problem size used to collect these figures is very small, which allows the entire dataset for each node to fit in the processor cache. It is likely that the performance for a larger per node problem size that does not fit in the cache would be much worse, since memory bandwidth, not processor performance, would be the limiting factor. Cache misses have a significant detrimental effect on performance for lattice QCD codes, however careful use of the cache pre-fetch capabilities of modern processors can largely eliminate these effects. Another disadvantage of the SGI Altix is its cost; a figure of €5 per MFLOP is quoted by Wettig which is substantially higher than competing systems [Wettig '05].

IBM's BlueGene system consists of a large number of processing nodes connected in a 3-dimensional toroidal network with nearest neighbour communications [A. Gara '05]. Each node consists of an ASIC chip with two IBM PowerPC 440 processor cores, with two floating point multiply-accumulate units attached to each one. The processors share access to memory and to the communications network. The system can run in one of two modes: co-processor mode, where one processor handles

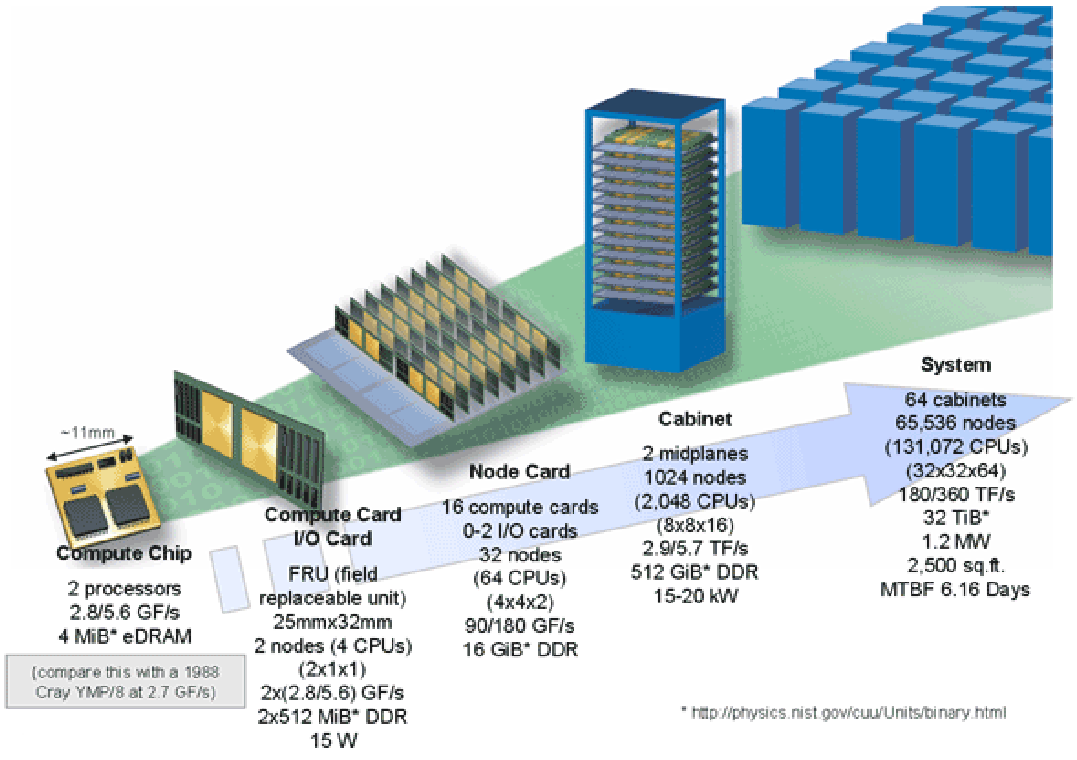


Figure 3-f. The construction of a large BlueGene/L supercomputer

communication and the other computation, and virtual-node mode where both cores are used for communication and computation. The virtual-node mode has twice the peak performance of the co-processor mode; however communication cannot be parallelised with computation as effectively for this mode.

BlueGene sustains about 20% of peak performance for the Dirac operator, when run in virtual-node mode, which translates to 1.12 GFLOPS per node, or about 560 MFLOPS per CPU core [Bhanot '05]. Price information for BlueGene systems is not publicly available however it is estimated that BlueGene has a price to performance ratio of around €2 per MFLOPS [Wettig '05].

3.2.2 PC Clusters

In the last few years clusters of PCs have become a significant source of computing power for high performance computing. In particular a significant amount of research effort has been directed at optimising PC clusters for lattice QCD [Wettig '05], [Gellrich '03], [Holmgren '06]. This effort has been very successful and there are now PC cluster machines, consisting of hundreds of nodes, dedicated to lattice

QCD, which can sustain over 650 GFLOPs for the core Dirac operator.

One of the key developments in applying PC clusters to lattice QCD has been the use of the SIMD processing capabilities of the latest PC processors. The SIMD instructions operate on short vectors of floating point numbers performing 4 single precision, or 2 double precision, operations per cycle, and have enabled dramatic improvements in performance. Luscher proposed this approach and published performance results for a single 1.5 GHz Intel Pentium 4, using a small problem size with single precision arithmetic and a single PC, of 1.5 GFLOPS [Luscher '02]. In comparison Gottlieb published results in the previous year showing performance of 186 MFLOPS, at single precision, for a 533 MHz Pentium III under similar conditions [Gottlieb '01]. Thus the use of the SIMD extensions, as proposed by Luscher, delivered a three-fold improvement in performance per MHz.

Generally the performance of PC clusters for single precision arithmetic is much better than for double precision arithmetic. There are two reasons for this: the SIMD pipelines can process twice as many single precision operations per cycle compared with double precision operations, and memory bandwidth requirements for double precision are twice that of single precision. For a significant problem size, where the dataset is too large to fit in the processor’s cache, PC processor performance is limited by memory bandwidth and not floating point processing capabilities [Wettig '05].

The end users of lattice QCD *prefer* to run their calculations using double precision since it gives more accurate results, however it is possible to run a calculation in single precision. Double precision is regarded by lattice QCD users as being safer at the simulation parameters that are more interesting from a scientific perspective. The penalty for using single precision, instead of double is that it introduces uncertainty about the accuracy of the simulation result. For PC based platforms single precision is usually twice as fast as double precision since memory bandwidth usually dominates the speed of calculations on these machines. Single precision is usually used on PC clusters since it allows a given simulation to be completed in half the time, when compared with double precision, or alternatively the number of sites in the lattice can be increased to take advantage of the extra performance.

Algorithmic improvements, when combined with improvements in processor speeds, memory buses, and peripheral buses and interconnect technologies, have dramatically improved the performance of PC based lattice QCD machines in recent years. PC based machines can now sustain around 2 GFLOPS for a single processor working on its own, and around 1 GFLOPS per processor in clusters of over 500 processors. PC clusters are now very price-competitive for smaller installations of less than around one thousand processors, with a price performance ratio of approximately $1 per MFLOPS [Wettig '05].

The challenge facing PC cluster designers now is not improving per node performance, but improving the performance of interconnection systems. Interconnects for lattice QCD clusters must have low latencies. Systems that use high latency interconnects have poorer per node performance than systems that use low latency interconnects. Lattice QCD machines send many small messages which makes latency more important than bandwidth. PC clusters are constructed using either Gigabit Ethernet [IEEE '99], which is low cost but has a high latency, or a more specialised interconnect, such as Infiniband [Infiniband '06], which has much lower latency but is significantly more expensive. Infiniband gives better per node performance, but the low cost of Gigabit Ethernet means that more processing nodes can be purchased for a given amount of money, which can compensate for the poorer per node performance of Gigabit Ethernet, based systems.

Some groups are looking to optimise PC interconnect technology to the needs of lattice QCD. The apeNET project, associated with the apeNEXT group, has developed an FPGA based interconnect system with very low latency targeted specifically at lattice QCD PC clusters [Ammendola '05].

3.2.3 Custom ASIC based machines – QCDOC & apeNEXT

QCD researchers are now reaching the limits of the "quenched" approximation, discussed in section 3.1.2. The quenched approximation ignores gluon to gluon interactions, and only simulates quark to gluon interactions, however the quenched

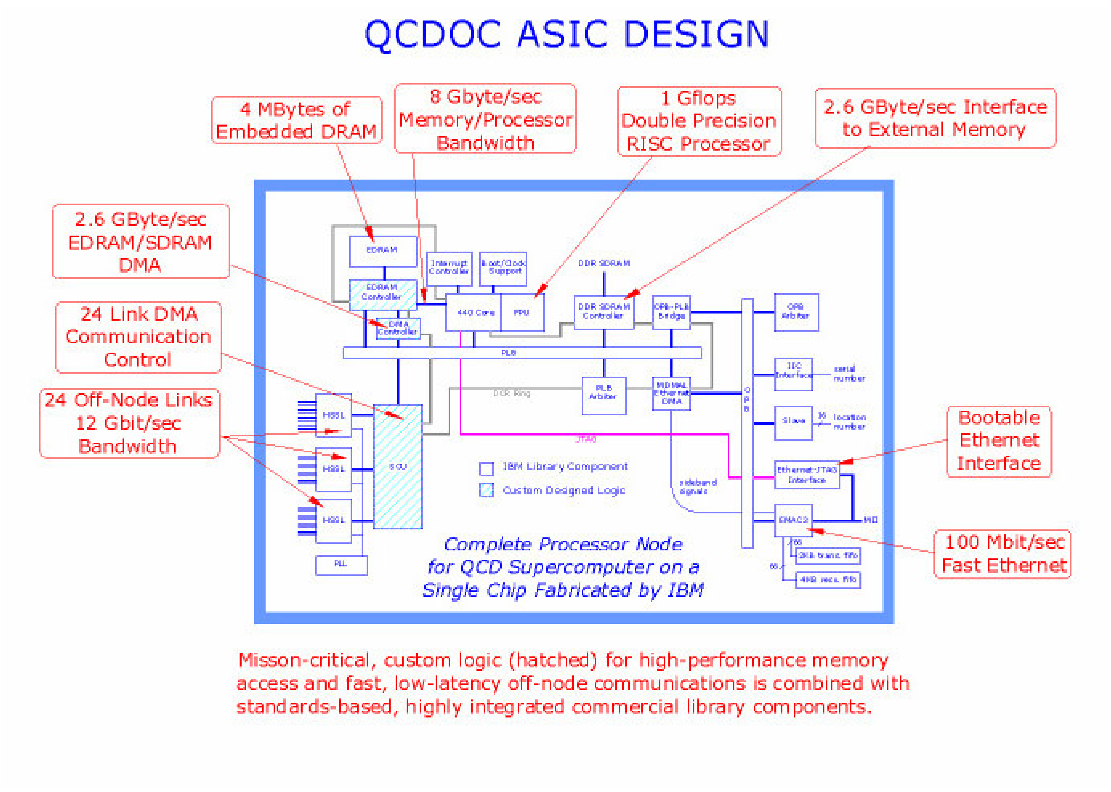

QCDOC ASIC DESIGN
4 MBytes of Embedded DRAM
8 Gbyte/sec Memory/Processor Bandwidth
1 Gflops Double Precision RISC Processor
2.6 GByte/sec Interface to External Memory
2.6 GByte/sec EDRAM/SDRAM DMA
24 Link DMA Communication Control
24 Off-Node Links 12 Gbit/sec Bandwidth
Bootable Ethernet Interface
100 Mbit/sec Fast Ethernet
EDRAM
EDRAM Controller
DMA Controller
Interrupt Controller
Boot/Clock Support
440 Core
FPU
DDR SDRAM
DDR SDRAM Controller
OPB-PLB Bridge
OPB Arbiter
PLB
DCR Ring
PLB Arbiter
MDMAL Ethernet DMA
OPB
I2C Interface
serial number
Slave
location number
JTAG
HSSL
HSSL
HSSL
SCU
PLL
IBM Library Component
Custom Designed Logic
sideband signals
Ethernet-JTAG Interface
EMACS
MII
2KB trans. fifo
4KB recv. fifo
Complete Processor Node for QCD Supercomputer on a Single Chip Fabricated by IBM
Misson-critical, custom logic (hatched) for high-performance memory access and fast, low-latency off-node communications is combined with standards-based, highly integrated commercial library components.

Figure 3-g. The design of a single QCDOC processor

approximation can only be used to investigate the heaviest particles. Lighter particles, such as protons, can only be investigated using full QCD where the gluon to gluon interactions are included.

Full QCD requires very significant computing resources however, which can only be met by a new generation of massively parallel computing machines capable of performing lattice QCD calculations at the rate of several teraFLOPS, or 10^{12} floating point operations per second. Two research collaborations, QCDOC [P. A. Boyle '05] and apeNEXT [Belletti '06], have produced massively parallel machines targeted at lattice QCD calculations to meet this requirement for teraflops scale computing power.

QCDOC

QCDOC machines are massively parallel machines, consisting of up to 12,288 processing nodes which can all be used on a single lattice QCD simulation. QCDOC stands for QCD On-Chip, since each processing node consists of a single chip into which all the components of the node are built. This chip contains all the components of the processing node, including the processor and the network controller. The chip contains a PowerPC 440 processor, with an attached floating point unit, along with a custom communications system.

The communications system consists of 12 bi-directional serial communications links which can connect the processing node to its 12 nearest neighbours in a 6-dimensional toroidal network. This system provides significant levels of bandwidth between large numbers of nodes along with a very low latency interconnect, which is critical for performance of lattice QCD codes. It is this sophisticated communications system that allows QCDOC machines consisting of over ten thousand processors to be built. Figure 3-g shows the structure of a single QCDOC processor including the off-chip communications systems, floating point processing units and the off-chip memory interfaces.

The individual QCDOC processors have relatively modest performance. They are currently running at 450 MHz and return sustained per node performance of 396 MFLOPS for the Dirac operator, which is the most computationally expensive part of lattice QCD. Consequently the QCDOC machines deliver their performance by applying a massive number of nodes to a single problem, and not through the individual performance of the processing nodes.

Two separate QCDOC systems of 12,288 nodes each have been installed in the Brookhaven National Laboratory, in the USA. Each of these systems is capable of sustaining nearly 5 TeraFLOPS on a single lattice QCD simulation. Another similar machine has been installed in the University of Edinburgh. Between them these three machines, along with the apeNEXT installations are enabling physicists to run massively more detailed simulations than ever before, which is testing QCD theory more and more intensively. QCDOC is also a very cost effective platform, since it costs around $1.1 per MFLOP. This is a similar cost to PC cluster based machinery, however the running costs of QCDOC are much lower with power consumption of less than 10 Watts per processor, including memory and communication systems. In comparison a PC based machine would consume at least 100 Watts per processor

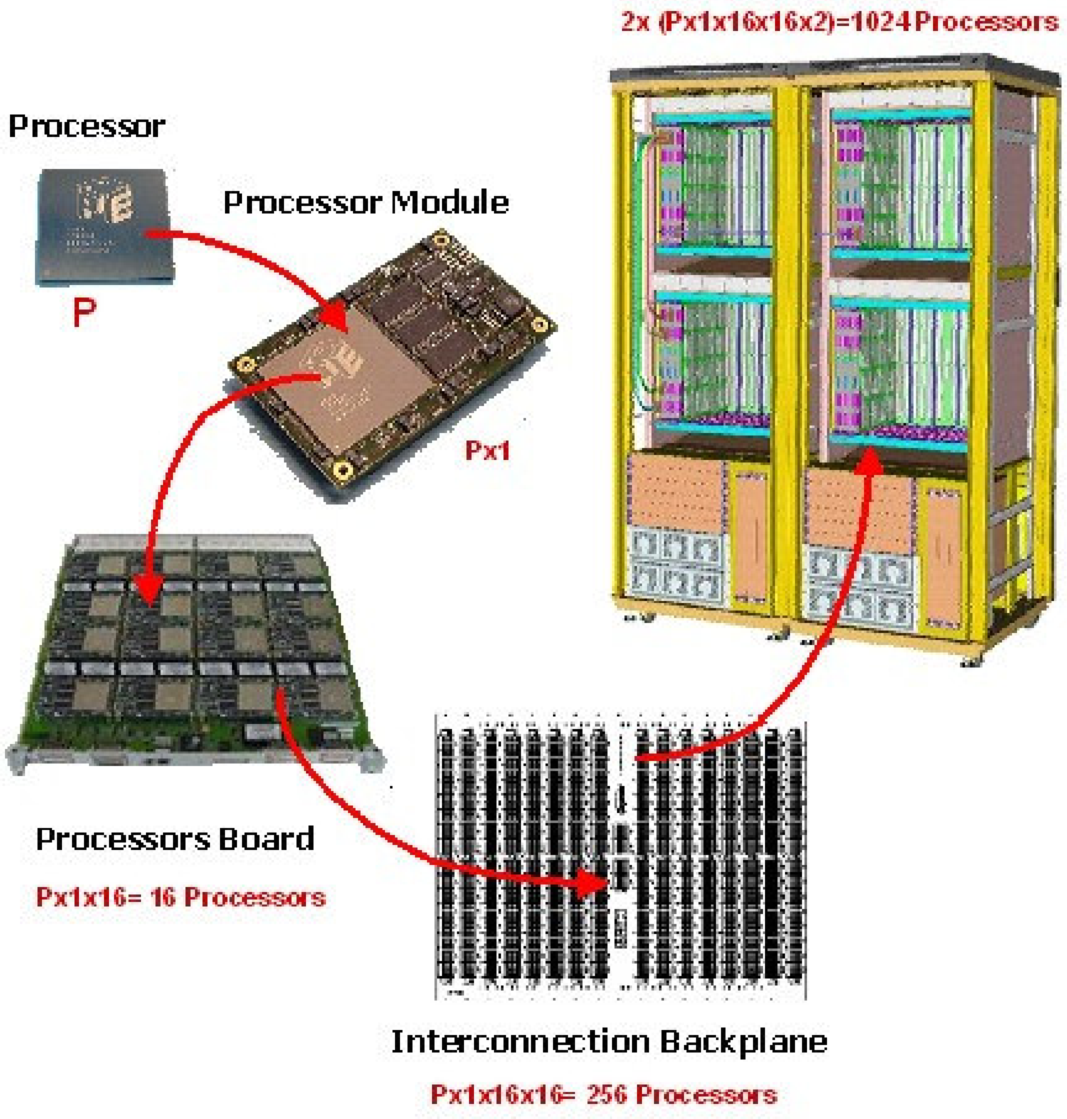

2x (Px1x16x16x2)=1024 Processors
Processor
P
Processor Module
Px1
Processors Board
Px1x16= 16 Processors
Interconnection Backplane
Px1x16x16= 256 Processors

Figure 3-h. The structure of a large apeNEXT machine.

(including communication and memory). This makes the QCDOC machine substantially cheaper to run [Wettig '05].

ApeNEXT

The apeNEXT machines are the latest in a series of machines from the APE (Array Processor Experiment) collaboration. In a similar fashion to QCDOC, the aim of the apeNEXT machine is to place all of the functionality of each processing node onto a single custom ASIC chip. These processing nodes are then connected up in large clusters, which can be dedicated to lattice QCD simulations. The designers of apeNEXT took a different approach to the QCDOC machine by giving each processing node significant floating point processing capabilities. The apeNEXT processors can perform eight floating point operations per cycle by implementing a complex-number multiply accumulate operation in hardware, which is the most common and computationally expensive operation in lattice QCD codes. The arithmetic unit is designed so that it can also perform other operations such as additions and subtractions, albeit with lower efficiency. ApeNEXT nodes can sustain performance of 896 MFLOPS for the core Dirac operator from a clock rate of only 160 MHz by using this complex number multiply accumulate unit.

Each processing node has 12 uni-directional communications links, which allow the node to be connected to all of its nearest neighbours in a 3-dimensional toroidal network [Belletti '06]. The links have very low latencies and good bandwidth; messages in lattice QCD are small, so low latency is far more important than high bandwidth. The high per-node performance of the apeNEXT system means that apeNEXT systems are smaller than the QCDOC systems at about three thousand nodes. Consequently interconnect performance is not as critical to overall system performance as it is for the QCDOC system.

To improve performance the processors can pre-fetch data from main memory. Data access patterns are very predictable in lattice QCD, so all application data can be pre-fetched, eliminating delays caused by the high access latencies of main memory.

ApeNEXT's combination of a low latency interconnect network and memory pre-fetch capabilities, when combined with the customised arithmetic of the individual nodes, allows each apeNEXT node to return excellent performance within systems consisting of several thousands nodes. As of late 2006, several apeNEXT systems are planned, or already installed, including a 6,656 node system for the National Institute of Nuclear Physics in Rome, Italy. The sustained performance of this system for lattice QCD calculations will be nearly 6 teraflops, all of which can be brought to bear on a single simulation. This system will enable simulations of far higher detail than has previously been possible.

The apeNEXT system delivers performance at a cost of about 1.2 Euro per MFLOP. This is comparable with the cost of both QCDOC and PC clusters. The apeNEXT processor consumes about seven Watts when running under load so it benefits from significant reductions in running costs when compared with PC based clusters.

3.3 IEEE Floating Point

This section describes the history and development of the two arithmetic systems that are used in this project; log arithmetic and IEEE double precision floating point.

3.3.1 Early History

IEEE standard 754 defines a standard approach for performing arithmetic on real numbers [IEEE '85]. Prior to the introduction of the standard there was many different systems for handling floating point arithmetic, which are detailed in an interview with W Kahan, conducted by C Severance [Severance '98]. The IEEE floating point arithmetic standard is heavily inspired by Kahan's work on computer arithmetic. Kahan was heavily involved in the standardisation process and was ultimately awarded the Turing prize for his work on floating point [Hennessy '90].

Different systems had different levels of precision, different ranges of represent-able numbers and each used its own rounding system. These differences were minor compared with the difficulties faced by programmers of some earlier floating point systems. For example many systems could return zero for X minus Y when X and Y were different. One system had numbers that behaved as non-zero numbers for addition and subtraction but as zero for multiplication and division. On another system some numbers, when multiplied by one, would result in an overflow, even though the number should not have changed.

The critical problem however was that, since each computer manufacturer had their own approach to floating point, the same code would return different results when run on different computers. Consequently it was becoming more and more expensive to develop reliable floating point domain applications since programmers had to spend inordinate amounts of time ensuring that their code ran correctly on any architecture it was used on. The IEEE standard for floating point arithmetic gives a

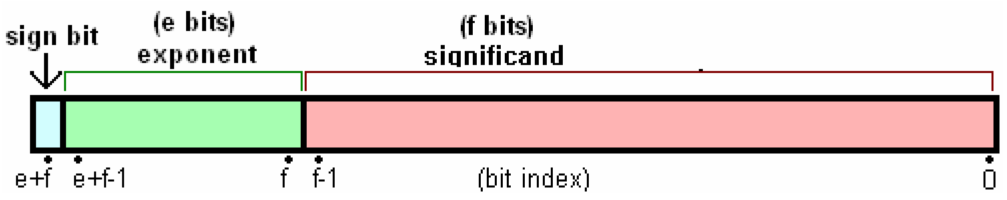


Figure 3-i. Structure of the IEEE floating point word

standard approach to floating point, including precision, range, rounding modes, exception handling, overflow and underflow. Nearly all modern processors conform to this standard, or at least to most of the standard, giving much more consistent behaviour across compliant processors.

3.3.2 The IEEE Floating Point Standard

IEEE floating point is analogous to the scientific notation commonly used to represent very large or very small numbers, for example, in scientific notation, the speed of light would be written as 2.99792×10^{8} ms^{-1} with 6 significant digits. Floating point numbers are in base 2, not base 10, so the speed of light written in binary scientific notation would be $1.0001110111100111100 0010 \times 2^{28}$ ms^{-1}. There are three parts to the scientific notation; the significand, the exponent and the sign. For the base 10 representation of the speed of light shown above, the significand is 2.99792, the exponent is 8 and the sign is positive. Similarly floating point representations have three parts, the sign, the exponent and the significand.

The terms significand and mantissa are often interchanged, however the IEEE standard recommends the use of the term significand when referring to the fractional part of a floating point number. This is because the term mantissa has a pre-existing mathematical meaning referring to the fractional part of a logarithm. Thus the IEEE prefers the term significand to avoid confusion. However designers of early implementations of floating point used the term mantissa to describe the fractional component of a floating point number, and this usage has persisted.

Figure 3-i shows the structure of a floating point word when stored in computer memory. The total number of bits in the word is $e+f+1$; f is the number of bits used for the significand, e is the number used for the exponent and a single bit is used for the sign. The value of *f* determines how precise the format can be, whilst the *e* determines the range. Larger values of *f* give more accurate calculations and larger values of *e* allow larger and smaller numbers to be represented. For IEEE single precision *e* is 8 and *f* is 23 and for IEEE double precision *e* is 11 and *f* is 52.

The significand is always normalised so that the first digit is always one (except in the case of de-normalized numbers which are discussed later). To increase precision this leading one is not stored in the floating point word, but is added automatically by the arithmetic units. Only the remainder of the significand to the right of the binary point is stored.

The value of the exponent can be positive for numbers larger than one or negative for numbers smaller than one. The value of the exponent is stored using a *biased* representation, where the actual value of the exponent has a bias added to it before storage. For single precision the bias is 127 and for double precision it is 1023. Thus for single precision an exponent value of zero is stored as 127 (in binary), an exponent of -43 would be 84 and an exponent of 110 would be 237. The *biased* representation is used instead of a format with an explicit sign because it simplifies the exponent comparison required for the addition operation. Finally a single bit stored in the most significant bit position of the data word specifies the sign of the number, zero for a positive number one for a negative number.

3.3.3 Early Implementations of Floating point on FPGAs

Fagin and Renard detail a very early implementation of IEEE single precision floating point on FPGAs [Fagin '94]. Fagin and Renard implemented a single precision adder and a single precision multiplier over four Actel A1280 FPGAs. Their adder used a three stage pipeline and could run at 4 MHz. Meanwhile their multiplier was not pipelined and required six cycles to produce a single result. Fagin and Renard also present a comparison of the resources required for arithmetic units to handle the rounding modes of the IEEE standard including handling de-normalised numbers. It is clear from the results presented by Fagin and Renard that, in 1994, FPGAs were not yet a viable platform for floating point arithmetic units. The FPGA based floating point units did allow the authors to explore interesting design tradeoffs with regards to the cost of including the various rounding modes and handling of de-normalized numbers.

Shirazi et al took a different approach to tackling the lack of resources on the FPGAs available at the time [Shirazi '95]. They described an implementation of 16-bit and 18-bit floating point arithmetic units for a Xilinx 4010 FPGA. These formats are considerably smaller than the single (32 bit) and double (64 bit) precision formats of the IEEE standard. The small formats allowed a single FPGA to host one of these units. The 16-bit and 18-bit formats provide about half the range and precision of single precision floating point, which rules them out for many calculations, but they provide a possible alternative to fixed point arithmetic which is commonly used for digital signal processing applications. The authors determined that one single precision floating point multiplier would need the resources of two of the Xilinx 4010 FPGAs used by the authors for their half precision implementations. Consequently FPGAs were still not a viable platform for floating point calculations.

Louca et al described an implementation of an IEEE single precision adder and multiplier [Louca '96]. Their adder is fully pipelined, and their multiplier uses a digit-serial multiplication to multiply the mantissas. As a result the multiplier requires 12 cycles to produce a single result. Implementing the integer multipliers required for floating point multiplication uses a lot of FPGA resources. Consequently a number of options are described for performing digit-serial multiplication of the mantissas and the resource requirements for these options are presented. It is clear from Louca et al, and from the multi-cycle multiplier described by Fagin and Renard, that the sheer size of the significand multipliers was a major obstacle to performing floating point arithmetic on FPGAs. The FPGAs available were too small to fit the 24 by 24 bit multiplier required for single precision floating point onto a single FPGA.

Ligon et al described FPGA implementations of a fully pipelined adder, and for the first time, a fully pipelined multiplier [Ligon '98]. The fully pipelined multiplier was made possible by the availability of larger FPGAs, combined with the use of a fully pipelined digit-serial integer multiplier for multiplying the mantissas. Ligon at al described a number of options for performing the mantissa multiplication, including fully pipelined digit-serial multipliers, bit-serial multipliers and booth recoding multipliers. They also presented multi-cycle bit-serial and booth recoding multipliers. Ligon et al showed that floating point on FPGAs was becoming a viable possibility however the resources required for the mantissa multiplication were still a significant problem.

Belanovic and Leeser published details of a parameterised floating point arithmetic library, and illustrated its use with an imaging algorithm [Belanovic '02]. They published performance and area results for a number of floating point formats, from an 8 bit format which uses 2 bits for the exponent and 5 bits for the significand, to an IEEE single precision equivalent format consisting of 32 bits, with 23 bits for the significand and 8 bits for the exponent. Belanovic and Leeser used a 12 bit wide version of the floating point cores to implement a K-means clustering algorithm which is applied to some satellite imagery. The implementation is a hybrid approach where some of the algorithm is performed in floating point and some in fixed point. They demonstrated satisfactory performance for this algorithm using their floating point cores.

Using non-standard floating point formats allows the arithmetic to be tailored to the numerical needs of the application, which can be useful in application fields such as digital signal processing or imaging. The numerical requirements of such fields can often be met using fixed point arithmetic; one example is Belanovic and Leeser's FPGA implementation of an imaging algorithm [Belanovic '02]. However only a few scientific applications can be run on single precision equivalent arithmetic; most require at least double precision equivalent arithmetic. Consequently small customisable floating point formats that are smaller than single precision are generally not useful for scientific applications.

3.3.4 Current State of the Art for Floating Point on FPGAs

The Impact of the Xilinx Virtex-II

The release of the Xilinx Virtex II FPGA [Xilinx '05a] was a major breakthrough for floating point on FPGAs. The Virtex II was the first FPGA to include hardware multipliers as part of the FPGA fabric. These multipliers can be used for integer multiplication inside an FPGA design. The multipliers can multiply two 18-bit integers in a single cycle at clock rates of over 100 MHz for even the slowest speed grade Virtex II FPGAs. Details of a number of floating point cores that take advantage of these hardware multipliers have been published in the literature. These cores are discussed in this section.

Roesler and Nelson published details of a number of optimisations for floating point units implemented on FPGAs [Roesler '02]. They were specifically interested in optimisations that make use of the Virtex-II hardware multipliers. Roesler and Nelson used the hardware multipliers to perform the significand multiplication required in floating point multiplication. Their results show a 77% reduction in resource requirements using this method when compared with a floating point multiplier that used a significand multiplier constructed from conventional FPGA resources. In addition Roesler and Nelson propose the use of the hardware multipliers to divide the significands in the floating point division algorithm.

Performance and resource utilisation results are presented for a variety of floating point formats, including IEEE single precision format. In common with other published work, which is discussed later in this chapter, Roesler and Nelson published results for non-standard floating point formats. The idea is that the floating point format can be tailored to better suit the numerical requirements of an algorithm, and so reduce the resource requirements of a design for that algorithm. Results are presented for arithmetic units that take advantage of the Virtex-II's novel architectural features. Results are also presented for units that are implemented using only basic FPGA resources, such as flip flops and look up tables.

The use of these units for a matrix based heat transfer application is also described; however no firm performance data is presented. The solution is based around a processing element that performs all processing required for the application. The authors speculate that 13 of these processing elements would fit on a six million gate equivalent Virtex-II FPGA. They predict performance of 2.2 GFLOPS, at single precision, for such a design. This, however, appears to be only an educated guess, and it is not clear whether this figure could be obtained for an actual implementation. If it were achievable then it would be an impressive result for an FPGA at that time.

IEEE Double Precision Arithmetic Becomes Possible

Prior to the introduction of the Xilinx Virtex-II, with its hardware multipliers, double precision arithmetic was not really viable on FPGAs. The significand multiplier for double precision was simply too large. FPGAs were available that could support a small number of floating point units, however their performance was poor. The limited availability of floating point units on these FPGAs restricted parallelism, and the achievable clock rates were very low. The Virtex-II was the first FPGA that could fit a substantial number of double precision units on a single chip, which, combined with the higher clock rates that the Virtex-II family FPGAs are capable of, made FPGAs a viable double-precision floating point platform.

After the introduction of the Virtex-II, FPGA based double precision floating point units began to appear. One of the first examples of these double precision units was published by Govindu et al [Govindu '02]. Govindu et al published area and performance data for both single and double precision, IEEE type floating point units and they presented a preliminary analysis of the double precision units when applied to a basic matrix multiplication application. The matrix multiplication algorithm is performed using specially designed processing elements. These elements perform the necessary steps for a matrix multiply on an FPGA, with each element using a single adder and a single multiplier. Performance data of 7 GFLOPS is published for a design that uses 29 of these units on a single, very large, Virtex-II Pro FPGA. However it is unclear whether this design was actually implemented in hardware. Such a design would require substantial memory bandwidth if it were to be used on a large problem size, and this issue is not addressed in this paper.

Performance Trends for FPGAs

Underwood published a significant paper in early 2004 which investigated the development of floating point arithmetic on FPGAs and made predictions for the future performance of FPGAs for floating point arithmetic [Underwood '04a].

Underwood developed a fully IEEE compliant floating point library which he implemented on a series of example FPGAs. The series of FPGAs was chosen to represent the development of FPGA technology over a six year period. The series starts with the first FPGA capable of supporting a single double precision IEEE floating point unit, the Xilinx XC4085XLA. An example FPGA from each successive Xilinx model range is included. Starting with the oldest they are XC4000 series, Virtex, Virtex-E, Virtex-II and Virtex-II Pro. Underwood charted improvements in peak floating point performance for this series of FPGAs and used the results to predict the future performance of FPGAs. He also presents comparisons with the performance of commodity processors, by comparing the performance of three of the FPGAs with the performance of commodity processors contemporary to those FPGAs.

Underwood's results show that the peak performance of FPGAs at both single and double precision was, at the time of publication, better than commodity CPUs for all three principal operations (addition, multiplication and division). In fact according to the results, the peak performance of FPGAs for addition has been superior since about the year 2000. The results also show that FPGA multiplication performance reached parity with commodity processors by 2003 and they show the performance of FPGAs for division to be substantially better than division on commodity CPUs. Using these results to extrapolate future performance, Underwood went on to predict the future peak performance of both commodity CPUs and FPGAs. He concluded that, if current trends continue, then FPGAs will continue to gain a substantial performance advantage over commodity processors in the future.

Peak performance figures only show the raw processing power of a device however. For commodity processors it is very difficult to get sustained performance for real applications equal to the processor's peak performance. Some highly optimised dense matrix computational kernels can obtain significant proportions of peak performance; in the best cases proportions of peak of nearly 90% are possible. However for real world applications, with large datasets the obtainable proportion of peak generally drops to about 50%, and for some applications the obtainable performance is even lower, sometimes dropping as low as 5% of peak. Scientific applications usually have large datasets that are far too large to fit in the processors cache, which means that such applications are often limited by memory bandwidth.

Building on the work published in his earlier paper, Underwood, in collaboration with Hemmert, published a second paper in 2004 which investigated the sustained performance of FPGAs for some core basic linear algebra subroutine libraries [Underwood '04b]. These libraries are commonly known as BLAS libraries. BLAS libraries are used in many scientific computing applications, so the performance of these libraries for a platform is strongly indicative of a platform's performance for full scientific applications. Underwood and Hemmert chose to implement some of the most common BLAS operations for a series of FPGAs and FPGA development platforms, which are used to represent the development of FPGA technology over a period of six years. In similar fashion to Underwood's earlier paper [Underwood '04a], Underwood and Hemmert then used these past performance trends to extrapolate future sustained performance trends for both FPGAs and commodity CPUs.

Underwood and Hemmert implemented three operations from the BLAS library; dot-product, dense-matrix-times-vector multiplication, and dense matrix-times-matrix multiplication. The operations were implemented for the same five FPGAs as used in Underwood's earlier paper [Underwood '04a], which is described previously in this section. The performance data for the five FPGA chips quoted by Underwood and Hemmert assumes that all the pins on the FPGAs are available for accessing off-chip memory. This is unrealistic however, since in a real FPGA based system a significant number of FPGA pins will be required for non-memory related purposes. So to gauge the effect of limited memory bandwidth on FPGA based BLAS routines, the designs were also implemented on three FPGA development platforms. The memory bandwidth of these platforms is limited by their design, so by implementing the BLAS operations on these platforms the authors were able to make a more valid comparison between FPGA and commodity processor performance.

Underwood and Hemmert found that the performance of the FPGA using all pins for memory was limited by processing capability for all three operations, whilst the performance of the FPGA development boards was limited by memory bandwidth for the dot-product operator and by processing power for the dense-matrix-times-vector and dense-matrix-times-matrix operations. In comparison the performance of the commodity processors was limited by memory bandwidth for the dot-product and dense-matrix-times-vector operations and by processing power for the dense-matrix-times-matrix operation.

The analysis performed by Underwood and Hemmert showed the performance of FPGAs and FPGA based reconfigurable platforms, in 2003, to be superior to commodity processors for both the dot-product and dense-matrix-times-vector operations. It also showed that the performance of FPGAs and FPGA reconfigurable platforms to be nearly equal to that of commodity processors for the dense matrix multiply operation. The author's performance extrapolation predicts that the performance advantage of the FPGA and FPGA based reconfigurable platforms over commodity processors should continue to widen for both the dot product and matrix-times-vector operations. Meanwhile the performance of FPGAs for dense matrix multiplication was predicted to continue to match that of commodity processors.

Moloney et al published detailed resource utilisation and performance data for a library of fully IEEE compliant floating point cores [Moloney '04]. This paper is included here since the double precision cores used for the double precision lattice QCD designs described in this thesis are the cores described in Moloney et al's paper. The double precision lattice QCD designs are discussed in Chapter 6. The library provides both single and double precision formats and it implements the full IEEE standard [IEEE '85].

In this section I have described a number of papers that take advantage of the Xilinx Virtex-II FPGAs to improve the performance of floating point on FPGAs. The FPGA based floating point units all took advantage of the combination of the large size of the Xilinx Virtex-II FPGAs and the hardware multipliers that first appeared on the Virtex-II to show that FPGAs had become a potentially viable computing platform for applications requiring floating point arithmetic.

The results published by Underwood showing FPGAs outperforming commodity processors in terms of peak performance for the three principal arithmetic operators (addition, multiplication and division) [Underwood '04a], and the results published by Underwood and Hemmert showing FPGAs outperforming commodity processors for two important basic linear algebra subroutines [Underwood '04b] are particularly important. These two papers indicate that FPGAs have considerable potential for accelerating scientific computing applications. However neither paper shows whether or not the instruction level parallelism provided by FPGAs is exploitable for real scientific computing applications. The floating point performance of FPGAs demonstrated in Underwood's paper comes from having a large number of floating point units working in parallel at modest clock rates. Real scientific computing applications must be able to exploit this parallelism to have good performance on FPGA based platforms.

3.3.5 The Future of Floating Point on FPGAs

Architectural Improvements in the latest FPGAs

The Virtex-II made FPGA floating point arithmetic viable. However FPGA technology has advanced considerably since the release of the Virtex-II FPGA family. Since then Xilinx has released three generations of FPGA, the Virtex-II Pro, the Virtex-4 and the Virtex-5 [Xilinx '06a][Xilinx '06b]. Each successive generation allows higher clock rates than the previous generations, which for designs requiring floating point arithmetic translates into higher performance. For example the hardware multipliers in the fastest Virtex-5 FPGA can run at over twice the clock rate of the hardware multipliers on the fastest Virtex-II FPGA. The hardware multipliers are often a bottleneck to increasing clock rate in floating point FPGA designs, so this could have a significant effect on performance. Virtex-5 also includes a number of other architectural modifications, including diagonal routing, which aim to reduce routing delay, and so improve performance.

The Virtex-5 also includes a new type of hardware multiplier that multiplies a 25-bit number by an 18-bit number. This is an innovation targeted at reducing the resource requirements of Virtex-5 based single precision floating point multipliers. The multiplication of mantissas (24 bit by 24 bit integer multiplication) for single precision is usually implemented using four hardware multipliers, enabling a fully pipelined operation. The mantissas are multiplied in four chunks, one 18-bit by 18-bit multiply, two 18-bit by 6-bit multiplies and a 6-bit by 6-bit multiply. The results of these multiplies are then added together in a particular pattern. See [Koren '93] for details on this operation. By increasing the width of the hardware multipliers to 25-bits by 18-bits then the number of hardware multipliers required for a single precision floating point core is halved to two. Indeed since the second multiplier is a 24-bit by 5-bit multiply, it could also be reasonably efficiently implemented using LUT based multipliers. Consequently the resource requirements of single precision floating point are much lower on the Virtex-5 family than on previous FPGA device families.

Possible Future Architectural Improvements

Beauchamp et al investigated the potential for embedded floating point units on a hypothetical FPGA which could be used to implement FPGA designs requiring floating point arithmetic [Beauchamp '06a]. A diagram of the proposed architecture is shown in Figure 3-j which is taken from the published paper. This idea takes its inspiration from the inclusion of specialised hardware such as hardware multipliers and block RAM in many FPGAs and the inclusion of the PowerPC processors in the Xilinx Virtex-II Pro [Xilinx '05b] and Virtex-4 FX FPGAs [Xilinx '06a]. The authors gathered information about the silicon area and delay characteristics of the logic blocks of a Virtex-II Pro FPGA. They then used this information to construct models of two hypothetical FPGAs. One of these hypothetical FPGAs has a similar structure to a Virtex-II FPGA, with configurable logic blocks, or CLBs, which are the basis of any FPGA, combined with embedded multipliers and on-chip RAMs. The other design replaces the embedded multipliers with embedded double precision floating point units.

A number of common high performance computing kernels, including matrix multiplication, matrix times vector multiplication, dot-product, and a fast Fourier transform, were then implemented using double precision floating point arithmetic for both of the hypothetical FPGAs. These kernels represent a good cross section of typical high performance computing workloads. It was found that the designs

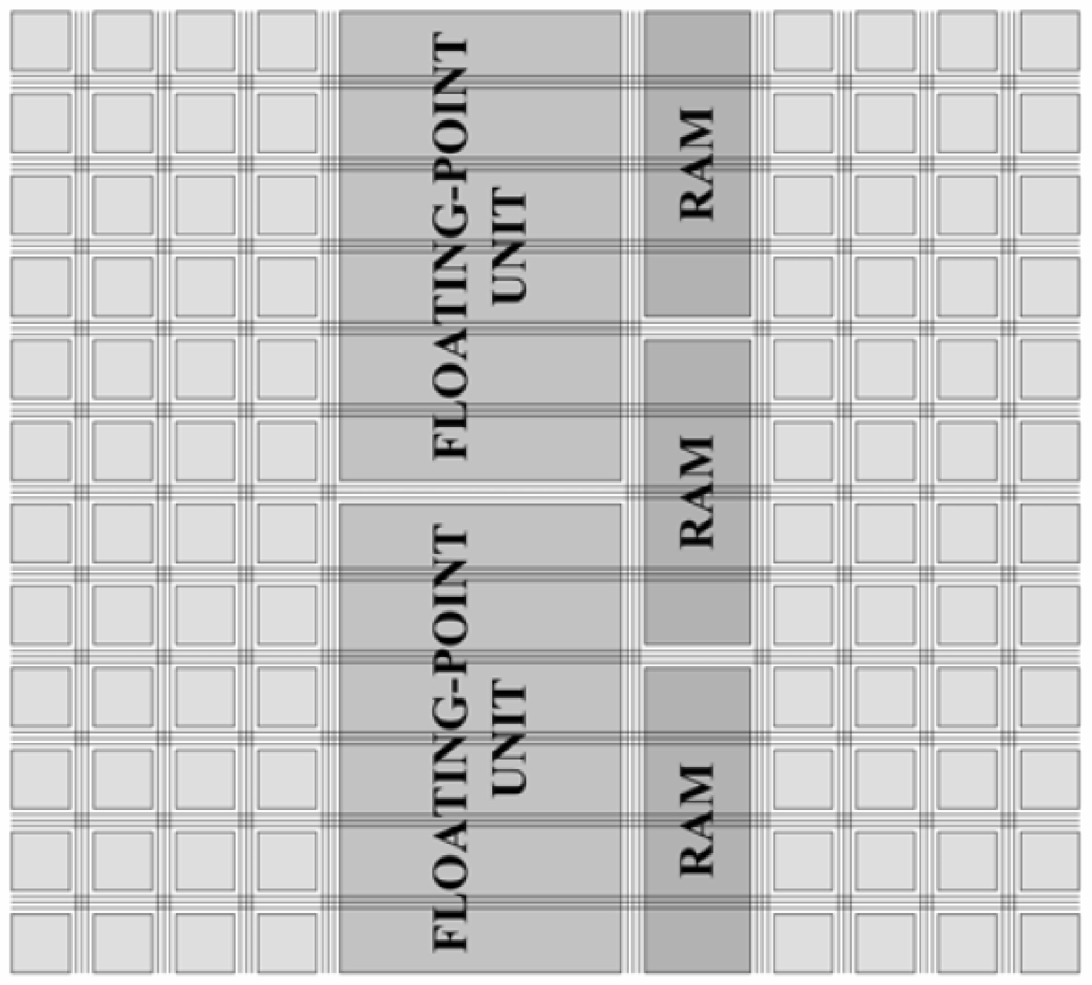


Figure 3-j. A diagram of the embedded floating point FPGA architecture proposed by Beauchamp et al [Beauchamp '06a].

implemented on the FPGA with embedded floating point units required, on average, 66% fewer slices than the same designs implemented on the FPGA with embedded multipliers. The authors also analysed the silicon area requirements and operating frequency of the two sets of designs and found that the designs implemented on the embedded floating point unit FPGA required, on average, 55% less silicon area and had clock rates 40% higher than the same designs implemented on the embedded multiplier FPGA. The authors conclude that an FPGA incorporating embedded floating point units would have very significant benefits for scientific computing applications that require double precision floating point arithmetic.

Unfortunately an FPGA incorporating embedded floating point units would only be suitable for applications requiring double precision floating point. The market for such an FPGA is small compared with other markets such as telecommunications, and consequently it is unlikely that such an FPGA will be produced in the near future. As a result of this constraint, Beauchamp et al proposed an alternative approach to embedding entire floating point units in an FPGA [Beauchamp '06b]. Floating point adders and multipliers both require barrel shifters to shift the significands during the calculation. Barrel shifters are normally implemented using the look up table (LUT) components of standard FPGA slices. Implementing shifters in this fashion requires a significant amount of FPGA resources, and make up nearly 30% of the adder and 25% of the multiplier used by the authors.

In order to reduce the resource requirements of FPGA based floating point cores, Beauchamp et al proposed two more efficient ways of implementing the barrel shifters required for floating point arithmetic. One approach is to embed shifter blocks into the FPGA fabric. This is a similar, but more general, concept to the embedded floating point concept proposed in their earlier paper [Beauchamp '06a]. Embedded shifters would only be useful for applications that had a significant requirement for such shifters. These shifters would be wasted for any application that did not require them.

In order to address this drawback the authors proposed including extra functionality in the basic configurable logic blocks of the FPGA. Each CLB in a Virtex-II type FPGA contains two flip-flops for storing data and two four-input look-up tables (LUT), which are used for implementing logic, including the variable shifters required for floating point . The authors proposed that two 4:1 multiplexers be added to the CLBs of an FPGA. The architecture of the proposed CLBs is shown in Figure 3-k, which is taken from the published paper. These multiplexers could then be used instead of the LUTs to implement logic such as barrel shifters. Barrel shifters implemented using multiplexers are much more resource efficient than those implemented using LUTs.

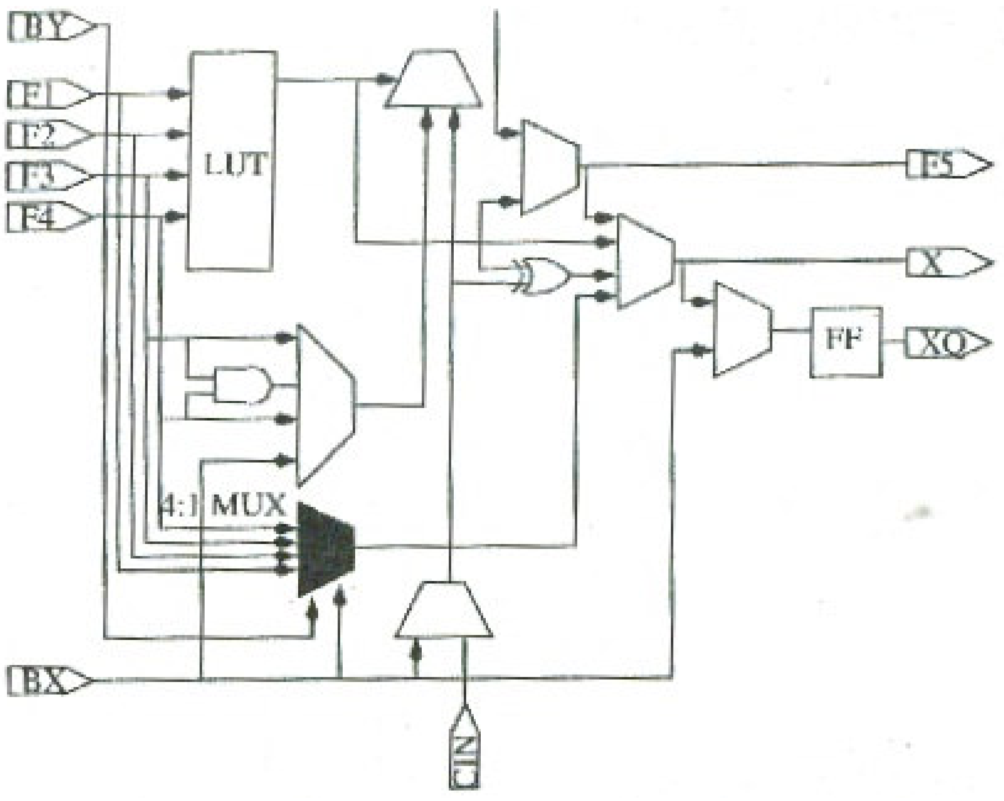


Figure 3-k. A diagram of the new CLB structure proposed by Beachamp et al [Beauchamp '06b]. A 4:1 MUX has been added to the CLB to improve performance of the variable shifters required for floating point arithmetic

Results presented for the two concepts show that both methods show a substantial reduction in FPGA resource requirements and increase in clock rate for a suite of typical high performance computing kernels. The inclusion of embedded shifters is shown to reduce resource requirements by 14.6% and increase clock rate by 3.3%. The inclusion of 4:1 multiplexers in the CLBs is shown to reduce resource requirements by 7.3% and to improve clock rate by 11.6%. Furthermore the use of the multiplexers was restricted to within the floating point units. Most FPGA designs include a number of substantial multiplexers, and the small 4:1 multiplexers could be used to implement any multiplexing logic within a design. Consequently the overall resource savings for an FPGA that includes multiplexers in the CLBs could be even larger than is suggested by the presented results.

3.4 Logarithmic Arithmetic

Logarithmic arithmetic is an arithmetic system that provides an alternative to conventional IEEE floating point for handling non-integer arithmetic. By representing numbers as fixed point logarithms, multiplication and division become very simple when using logarithmic arithmetic. In comparison IEEE floating point multiplication and division are more complex. However logarithmic arithmetic addition is more complex than floating point addition, and consequently a hardware implementation of logarithmic addition requires substantial hardware resources. Thus logarithmic arithmetic can be an excellent choice for algorithms which involve a significant amount of multiplication and division. This section describes the development of logarithmic arithmetic, from when it was first proposed as a method for handling non-integer arithmetic, up to the development of the FPGA arithmetic cores that are used in the lattice QCD designs described in Chapter 5.

3.4.1 Early investigations of logarithmic number systems

Some of the earliest work on logarithmic arithmetic was performed in 1962 by Mitchell, who described algorithms for computer multiplication and division which used binary logarithms [Mitchell '62]. Combet et al. expanded on Mitchell’s work and described a method of computing the base two logarithm of binary numbers [Combet '65]. The result was a method for handling binary non-integer arithmetic, which provided a simple fast way of multiplying non-integer numbers, along with a way of converting binary numbers to binary logarithms. Thus the method provided a way to multiply binary numbers by converting them to binary logarithms, multiplying them and then converting them back to binary numbers again.

Unfortunately the accuracy of this method was modest, which limited its application to fields such as digital filtering, where the accuracy of individual calculations is not important. In such fields it is the accumulated error over the entire calculation that matters most, and the lack of rounding error when multiplying binary logarithms means that logarithmic arithmetic has a very low accumulated error. Multiplication of log arithmetic numbers is performed by adding the two logarithms using an integer add operation. This operation never has a remainder, which means no rounding is required. Thus there is no accumulated rounding error for log arithmetic multiplication.

Several years later, in 1970, Hall et al described the application of binary logarithms to such digital filtering applications [Hall '70]. They concluded that the error rates of binary logarithms rule them out as an arithmetic system for general purpose computer applications. However they concluded that they could be used for applications that are less sensitive to high error rates, such as digital filtering.

3.4.2 Towards a hardware implementation

Building on the work of Mitchell and of Combet et al, Kingsbury presented a full logarithmic arithmetic system which was capable of performing all the basic arithmetic operations (addition, subtraction, multiplication and division) in the logarithmic domain [Kingsbury '71]. The significant contribution of this work is that it described a method for native addition and subtraction of binary logarithms, which eliminated the conversion steps that were required in earlier work. Kingsbury and Rayner proposed logarithmic arithmetic as an alternative arithmetic system for implementing digital filters, and they compared its performance with fixed point arithmetic, which is the conventional system for digital filters.

Fixed-point arithmetic is commonly used in digital filters and digital signal processing because it is a simple and fast way of handling non-integer arithmetic on computer systems which lack floating point processing capability. Fixed point arithmetic can only operate accurately on numbers that are within a limited range, which can cause poor performance for digital filters, because signals with small amplitudes are often lost after being passed through the filter. Log arithmetic was presented by Kingsbury and Rayner as an alternative arithmetic system which, owing to its substantially better dynamic range, could give much better performance for digital filtering algorithms. They proposed a 16-bit log arithmetic format which they calculated to have a dynamic range of 9000:1, which is far better than the dynamic range of 16-bit fixed point at 32:1.

Kingsbury and Rayner also described a storage format for logarithmic numbers, including how to deal with sign and zero. In order to represent a non-integer number, say x, in the logarithmic domain, the fixed-point value $i=log_2|x|$ is calculated. This gives a fixed point value, i, in the logarithmic domain which represents the value of the number. However the sign of the number is not represented in i since it is not possible to find the log of a negative number. Consequently the sign must be represented explicitly. Kingsbury and Rayner represented the sign by appending an extra bit to the start of the log value. This bit is set to zero for a positive number and to one for a negative number. Also since the log of zero is minus infinity, zero must also be represented separately. Kingsbury and Rayner proposed using a bit in the data word for this purpose, which is set if the number is zero.

A significant advantage of the proposed logarithmic number system is that multiplication and division are very simple and can be achieved by simply adding, or subtracting, respectively, the two binary logarithms involved in the operation. Unfortunately this simplicity comes at the price of complex and expensive addition and subtraction. The equation for logarithmic addition, shown below in equation (12), requires a method to calculate power to the base-2 as well as log to the base-2. Both of these functions are difficult to implement at high speed in hardware.

$$log_2(x+y) = i + log_2(1 + 2^{j-i}) \text{ where } i=log_2x \text{ and } j=log_2y \qquad (12)$$

Kingsbury and Rayner proposed two methods for solving equation (12). The first approach, which they call the direct method, involves solving equation (12) directly using dedicated hardware circuitry designed for the task. The other approach is the read-only memory method, which uses look-up tables of pre-calculated values to compute the result. The look-up tables are quite large however, so Kingsbury and Rayner proposed storing a subset of the tables, and then using interpolation to calculate the result. They concluded that this approach was likely to have the best performance for an actual hardware implementation. They also demonstrated a digital filter based on their logarithmic arithmetic system which has significantly better filtering performance than the same filter implemented using fixed point arithmetic.

In 1975 Swartzlander and Alexopoulos presented detailed algorithms for all the major log arithmetic operations [Swartzlander '75]. Their implementation of the addition/subtraction algorithm uses the look-up table method previously proposed by Kingsbury and Rayner [Kingsbury '71]. They concluded, given the state of the art for read only memory at that time, that the maximum word size for a hardware implementation of logarithmic arithmetic was twelve bits. This word size is small and meant that the proposed logarithmic arithmetic system was only suitable for a number of specialised fields including digital filtering and image enhancement. Thus at this stage, in 1975, hardware implementations of logarithmic arithmetic were substantially limited by the technology available.

Five years later, 1980, Kurokawa, Payne, and Lee analysed the error introduced for recursive digital filters when implemented using logarithmic arithmetic [Kurokawa '80]. They concluded that the signal-to-error ratio from LNS arithmetic was lower than that produced by conventional floating point arithmetic, assuming both systems use the same word size. These results demonstrated that the numerical capabilities of logarithmic arithmetic are at least the equal of the more conventional floating point arithmetic system.

Three years later two papers were published which took advantage of the numerical advantages of logarithmic arithmetic to implement a 2 dimensional digital filter and a fast Fourier transform (FFT) filter. Sicuranza [Sicuranza '83] used the logarithmic arithmetic system to implement 2-D digital filters. He also outlined a method to alter the range and precision of a logarithmic number system, which allows the system to be customised to the numerical requirements of a particular application. Meanwhile Swartzlander et al implemented a fast Fourier transform using logarithmic arithmetic which, owing to the superior numerical performance of the logarithmic system, returned better numerical performance than FFT filters implemented using either floating-point or fixed-point arithmetic [Swartzlander '83].

3.4.3 Hardware Implementations of Logarithmic Arithmetic

All of the work described in the previous section was implemented using software arithmetic systems; the operations were performed using software algorithms on a conventional processor. This was the most common way of implementing any non-integer arithmetic system at that time, since most processors lacked dedicated non-integer processing facilities. However the development of the IEEE standard for binary floating point arithmetic in 1985 signalled widespread availability of hardware-based floating point arithmetic on commodity processors. Thus software based logarithmic arithmetic systems no longer had sufficient performance to be competitive with the more widely used floating point arithmetic system.

Coleman et al proposed a logarithmic arithmetic microprocessor as part of the High Speed Logarithmic Arithmetic (HSLA) project which would provide a hardware implementation of 32-bit logarithmic arithmetic with equivalent numerical performance to single precision floating point arithmetic [Coleman '00]. This implementation took the form of a dedicated silicon chip with a peak performance of 650 MFLOPS at a clock rate of 166 MHz [Matousek '03]. However by the time this chip was developed commodity processors were capable of returning much better performance and the potential for a dedicated logarithmic microprocessor was limited. A major difficulty is the complexity and expense of the latest chip fabrication technologies which effectively rule out mass production of a dedicated logarithmic microprocessor.

FPGAs are an alternative platform for implementing dedicated hardware such as logarithmic arithmetic systems. FPGAs allow high speed implementations of hardware designs without the high costs and complexity of creating a dedicated silicon design. Matousek et al presented an FPGA based logarithmic arithmetic system which provides an alternative to conventional floating point arithmetic for FPGA designs [Matousek '02]. Their work is part of the HSLA project that produced the logarithmic microprocessor described previously in this section. This system is used to implement the logarithmic arithmetic designs described in Chapter 5.

The arrival of the Xilinx Virtex-II FPGA made FPGA based implementations of the logarithmic arithmetic system possible [Xilinx '05a]. The logarithmic adders in the HSLA project use a set of look-up tables to perform their calculation and these look-up tables are stored in the on-chip block RAM of the Virtex-II FPGA. Without these block RAMs, the FPGA implementation would not be possible. The HSLA logarithmic arithmetic units are described in the following sections.

3.4.4 HSLA Multiplication and Division

Floating point multiplication and division are difficult to handle efficiently in FPGA logic. The hardware logic required for multiplication and division of the mantissas consumes a lot of FPGA resources. The introduction of hardware multipliers on the Xilinx Virtex-II FPGA has made floating point multiplication much more practical since the hardware multipliers can be used to multiply the significands. However division still remains a problem. The resource requirements of a library of IEEE compliant floating point units developed by Underwood show that floating point adders and multipliers require a similar number of slices (the multiplier also requires a number of hardware multipliers), whilst fully pipelined division units require over four times as many slices as either the adder or multiplier [Underwood '04a].

The HSLA log arithmetic cores provide an alternative to high cost floating point multiplication and division. Multiplication and division in log arithmetic can be performed in minimal computation time and without rounding errors since they have the same level of complexity as a fixed-point addition. Assuming $i=\log_2|x|$ and $j=\log_2|y|$, the calculations are performed using equations (13) and (14).

$$\log_2(x \times y) = i + j \qquad (13)$$

$$\log_2(x / y) = i - j \qquad (14)$$

The multiplication operation is shown in equation (13) above, and shows that multiplying two binary logarithms is as simple as an integer addition. The sign of a logarithmic number is represented separately to the number, as explained in the previous section, section 3.4.3. However in a change to the scheme proposed by Kingsbury and Rayner [Kingsbury '71], a single special value, instead of a dedicated bit, is used to represent zero. Using a special value provides extra precision compared with the system proposed by Kingsbury and Rayner, since an extra bit in every data word is available for representing the binary logarithm.

As a result a small amount of extra logic is required to handle the sign bits and to check for zeroes in the calculation. However the multiplication operation is still very simple. In comparison floating point multiplication requires a large integer multiplier (for multiplying the significands) and o, which makes floating point multiplication much more complex than log arithmetic multiplication.

The division operation is shown in equation (14) above. It can be seen that log arithmetic division requires only an integer subtraction, along with logic to handle the sign bit and to check for zeroes. In comparison floating point division requires division of the significands and the circuitry for such a divider is very large and complex. The size of the floating point division circuitry gives log arithmetic a clear advantage over floating point division.

3.4.5 Addition and Subtraction

Unfortunately the small size and high efficiency of the log arithmetic multipliers and dividers comes at a price; addition is comparatively expensive. Once again, assuming $i = \log_2|x|$ and $j = \mathrm{log2}|y|$, addition may be evaluated as shown in (12). The equation for logarithmic addition requires a method to calculate power to the base-2 as well as log to the base-2. These are non-linear operators, and as such are difficult to implement efficiently. For this reason, all practical implementations use a Taylor series approximation in conjunction with a pre-calculated lookup value. The pre-calculated lookup values are stored in look-up tables, the size of which increases exponentially as the number of bits used in the log arithmetic word increases.

The sheer size of the look-up tables makes log arithmetic difficult to implement. In order to reduce the size of the look-up table's interpolation between values is generally used along with smaller look-up tables. However, these smaller tables come at the expense of accuracy. Coleman and Chester observed that *"this function is irrational and thereby subject to a half-bit rounding error, the interpolation procedure tends to introduce additional error, and the entire process is time consuming"* [Coleman '99]. They proposed a novel error-correction algorithm, which gives equivalent performance and accuracy to floating point at 32-bit precision. Their design involves the use of some additional lookup tables for interpolation, but crucially, the error correction steps may be performed in parallel with the rest of the

FPGA Component	**Normalised Area (1 unit = Area of 1 slice)**
Slice	1
18 kilobit block RAM	27.9
Hardware Multiplier	17.9

Table 3-a. Area in silicon of three FPGA components. One unit is the area required for a single FPGA slice.

calculation, thereby causing no degradation in performance.

3.4.6 Accuracy and Performance of the HSLA System

In July 2000, Coleman et al used detailed simulations to compare the error produced by 32-bit floating point against 32-bit LNS [Coleman '00]. The paper concluded that the logarithmic addition algorithm had substantially the same error as floating point. Coleman et al also presented a number of large scale case studies testing the effectiveness of the LNS system in three different numeric-intensive applications; digital signal processing, graphics, and numerical methods. The log arithmetic cores were used as if they were floating point cores; their use was not tailored to the specific advantages and disadvantages of LNS. This provided an objective, and hopefully realistic, test environment. All three cases described by Coleman et al show a substantial performance improvement for the LNS version, requiring between 41% and 69% of the original time to execute.

3.4.7 Comparing LNS and IEEE floating point

One of the main barriers to the acceptance of logarithmic arithmetic is the lack of a suitable metric for comparison with IEEE floating point. To address this problem Haselman et al presented a comparison of log arithmetic cores with IEEE floating point arithmetic cores. A system of "normalised slices" was used to compare the resource requirements of log arithmetic and floating point cores [Haselman '05]. By examining pictures of the silicon of a Xilinx Virtex II FPGA they calculated the relative area of three components of the Virtex-II architecture; 18-kilobit block

RAMs, hardware multipliers, and slices. The calculated values are shown in Table 3-a. See section 2.1 for an introduction to FPGA architectures.

This comparison metric allowed the authors to compare the resource requirements of floating point cores to the resource requirements of log arithmetic cores in two ways; by the number of units that a particular FPGA can support and by the number of “normalised slices” that each unit requires. Haselman et al go on to present an evaluation of what mix of arithmetic operations an algorithm requires for it to benefit from the use of log arithmetic instead of IEEE floating point arithmetic.

The authors determined that arithmetic operations required by an algorithm needs to consist of more than 70% multiplies by volume for a log arithmetic implementation to be smaller than a floating point implementation. However they found that if more than 45% of the operations on the critical path of the algorithm are multiplies then, owing to the shorter latency of the LNS multiplier, the performance of a log arithmetic implementation would be better than a floating point implementation. Thus the short latency of log arithmetic multipliers can offset the comparatively large size of the adders.

Interestingly, whilst the log arithmetic cores described by Haselman et al are quite different in construction to those used in this work, which were developed by Matousek et al, the normalised area for both systems adder pipelines is virtually the same. The Matousek cores require 1286 normalized slices per adder pipeline, and the Haselman cores require 1291 normalised slices per adder pipeline [Haselman '05][Matousek '02]. The Haselman cores use a large number of the on-FPGA hardware multipliers provided by Xilinx Virtex-II FPGAs and they use a comparatively small number of block RAMs. Conversely the Matousek cores use a large number of block RAMs and comparatively few hardware multipliers. Consequently the conclusions that Haselman et al draw for their logarithmic cores are equally valid for the Matousek cores used in this work.

3.5 Summary

This chapter introduced quantum chromodynamics (QCD) theory and explained why scientists must use computer simulations to test this theory. The use of computer simulations for this purpose, called lattice QCD, is the focus of a significant amount of worldwide research work. Part of this work focuses on developing computing machinery that is dedicated to the needs of lattice QCD simulations, including a series of PC clusters and two custom ASIC based supercomputers. In addition some groups have investigated the use of commercial supercomputers for lattice QCD. These machines provide a view of the state of the art of high performance scientific computing machinery, and by comparing the performance of the FPGA designs described in this thesis with these machines I can draw significant comparisons about the suitability of FPGAs for high performance scientific computing.

For several years FPGAs have been successfully used to implement applications that use integer or fixed point domain arithmetic. Recent developments in FPGA technology mean that FPGAs can now support a significant number of non-integer arithmetic units on a single FPGA. Consequently FPGAs are now a potentially viable platform for high performance computing applications. This chapter charts the development of the two non-integer arithmetic systems used in this work.

The first system, single precision equivalent logarithmic arithmetic, is an alternative to conventional IEEE floating point, which provides low cost multiplication and division by representing real numbers as fixed point logarithms. The development of the logarithmic arithmetic system is described up to the most recent FPGA based implementations of the system. The second arithmetic system is IEEE double precision floating point. This is the standard arithmetic system for high performance computing applications and is only recently available for FPGAs. Implementing floating point arithmetic for FPGAs has been the focus of substantial research over several years, and this research is described in detail. The following chapters describe the design and implementation of FPGA based lattice QCD designs using both single precision log arithmetic and double precision floating point arithmetic.

Chapter 4

Algorithm Analysis

Careful and extensive algorithm analysis is an essential step in developing a high performance implementation of any algorithm. This chapter presents a detailed and comprehensive analysis of the lattice QCD algorithms that are implemented in this thesis. The analysis is presented separately since it is applicable to the designs described in both Chapter 5 and Chapter 6. The algorithms analysed are: the performance critical Dirac operator, the dot-product operator and the matrix add-scale operator. These three operators are used to construct the conjugate gradient solver application. The analysis evaluates a number of constraints to performance, including memory bandwidth, maximum clock rate and exploitable parallelism.

4.1 Performance Metrics and Considerations

Performance of an FPGA design that requires non-integer arithmetic may be limited by a number of factors, including non-integer arithmetic performance, memory bandwidth requirements and the maximum clock rate of the designs. Performance in high performance computing machines is usually measured in floating point operations per second (FLOPS).

The number of floating point operations performed per second is determined by the number of floating point operations performed per cycle and by the system clock rate. Exploiting low level parallelism increases floating point operations per cycle. Efficient design and using highly pipelined arithmetic units improves clock rate. Unfortunately most applications involve dependant operations, where the result of one operation is the operand for another operation. Long pipeline latencies are a significant problem for such applications since the arithmetic pipelines may be forced to stall frequently while waiting for operands to be produced. Consequently there is a significant trade-off between parallelism and clock rate for many FPGA designs. The key is not to attempt to maximise either clock rate or parallelism but to maximise the product of the two.

The latencies of the arithmetic cores used for the designs in this thesis are fixed. The logarithmic cores are discussed in section 2.3.1, and the double precision floating point cores are discussed in section 2.3.2. Briefly, the logarithmic adder has a relatively long pipeline latency of 9 cycles whilst the multiplier has a very short latency of only one cycle. The cores can run at clock rates of around 90 MHz. Furthermore the logarithmic adder requires significant FPGA resources whilst the logarithmic multiplier requires very few. Consequently the availability and use of the adders is the performance constraint for designs using logarithmic arithmetic.

The double precision arithmetic units have latencies of 6 cycles for the adder and 7 cycles for the multiplier. These latencies are low for double precision floating point units, however as a result their maximum clock rate is also comparatively low. This, however, can be an advantage for many algorithms since short latencies make it easier to exploit parallelism for applications with dependent operations. Meanwhile the lower clock rate does not affect performance since it is difficult to get large complex designs such as those described in this thesis to run at high clock rates.

4.2 Clock Rate and Path Delay

A design's maximum clock rate is determined by the delay of the longest path in the design. Path delay has two components; logic delay and routing delay. Logic delay is determined by the complexity of the logic in the path. For example if two 64 bit numbers are added in a single clock cycle and then multiplied by a third number, as shown in Figure 4-a, a significant logic delay will result, leading to a slow clock rate. This delay can be reduced by pipelining the operation over two or more cycles, which is shown in Figure 4-b. Logic delay can also be caused by having long logic delays in the condition checks of if-statements and loops. The condition check must

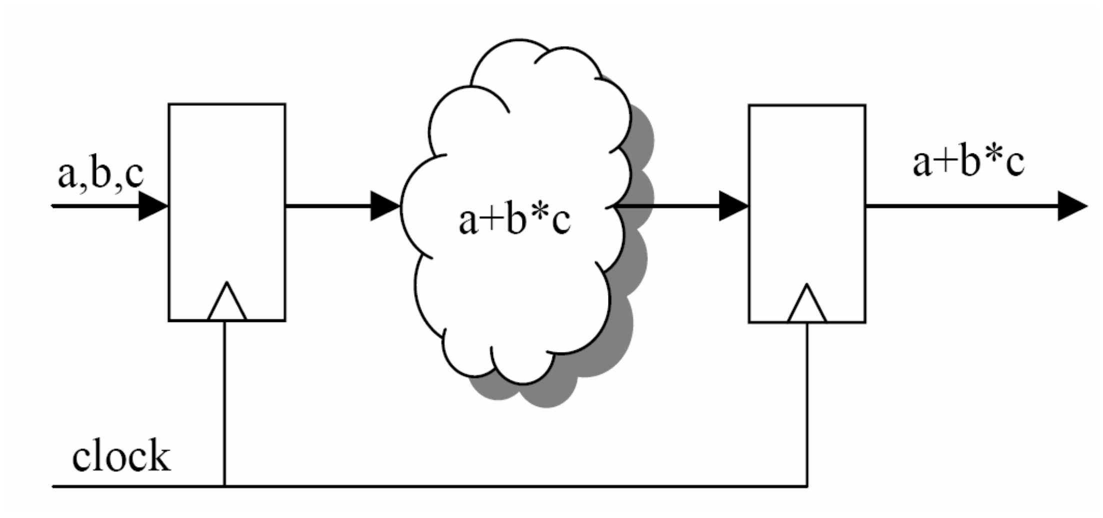


Figure 4-a. An example of an un-pipelined circuit with a large logic delay.

be evaluated in sequence with the expression, leading to long delay. It is usually possible to pipeline the evaluation of the condition check, thus reducing logic delay.

Routing delay is the delay caused by the wire delays in the internal wiring of the FPGA. Logic in an FPGA design is implemented using the internal logic blocks of the FPGA. The basic logic cells of the FPGA (see section 2.1) are very small and must be connected to together to construct the complex logic involved in an FPGA design. These connections are implemented using the internal wiring fabric of the FPGA. However this wiring causes a delay in the logic path and the longer the wiring the greater the delay. Long wire delays can be reduced by reducing sharing of on chip resources. For example if a component such as an IO pin is connected to a location on chip that is some distance away then a long wiring delay will be created. Heavy sharing of resources, such as arithmetic units, should be avoided since the multiplexer logic required to manage access to the resource adds significant delay to the logic path.

These are not the only considerations for a hardware designer, they are just some examples. Hardware design is complex and care must be taken to avoid introducing unnecessary delays into a design.

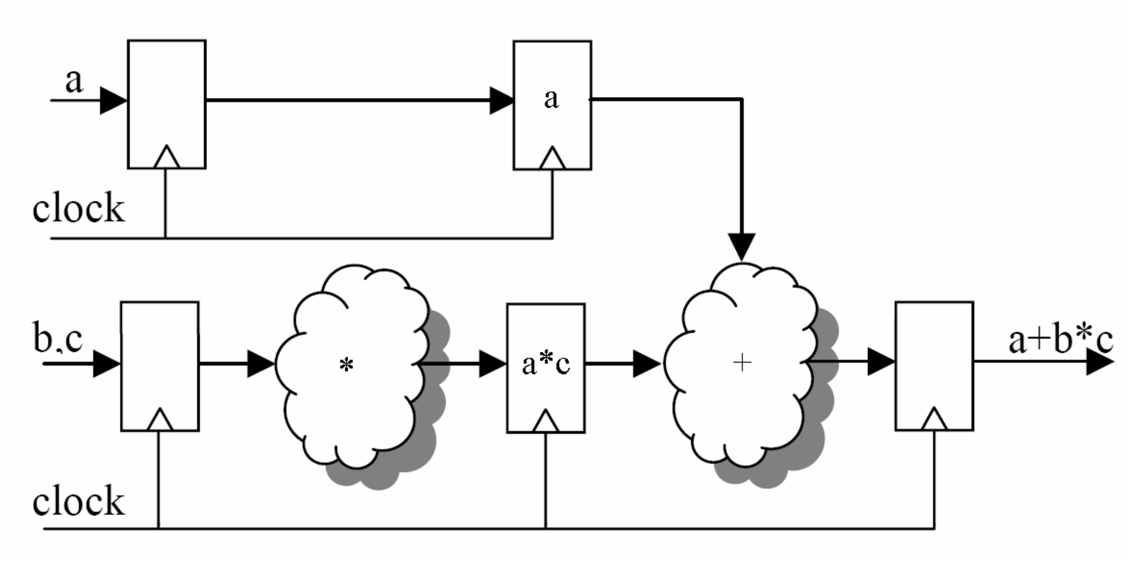


Figure 4-b Pipelined version of the circuit shown in Figure 4-a.

4.3 Exploiting Low Level Parallelism

FPGAs provide considerable instruction level parallelism, which can be exploited for a particular application by using a design that is customised to the needs of that application. Parallelism for a design requiring non-integer arithmetic is obtained by performing as many arithmetic operations per cycle as possible. The designs described in this thesis use multiple arithmetic units to exploit this kind of fine-grain parallelism.

Significant FPGA resources are required to implement the application logic required to control use of the arithmetic units. One cannot simply fill the chip with as many arithmetic units as possible and hope for the best, it necessary to consider a number of other factors, including the memory bandwidth requirements of the design. There is no point creating a design with excellent potential performance if the available memory bandwidth is insufficient.

Consequently the focus of the algorithm analysis was to establish the floating point arithmetic and memory bandwidth requirements of the algorithms. FPGAs provide a lot of low-level parallelism, but this parallelism can only be exploited for applications that have a high ratio of floating point operations to memory bandwidth, since performance is determined by the memory architecture of the system, and not

	Addition	Multiplication
Dirac Operator	1440	1176
Matrix Add Scale	24	24
Dot Product	24	24
Conjugate Gradient	3000	2472

Conjugate gradient also requires 2 divisions per main loop iteration.

Table 4-a. Arithmetic operations per site for lattice QCD operations

by the design the system runs. A low ratio means the application's performance will be memory bound, which limits the potential for exploiting instruction level parallelism for such a design.

4.4 Analysis Techniques

Most applications are heavily dependent on unpredictable inputs, often from the end user, which affect the computational load for the application. Thus it is often very difficult to analyse the algorithm with source code analysis alone. A common alternative analysis technique is to use profiling tools to measure the time spent in each part of the application. This information can then be used to assess where in the application the computational load lies.

The lattice QCD algorithms, in common with many matrix based scientific computing algorithms, are completely loop based. For example the core Dirac operator source code has no *if* statements or *while* loops of any sort making the computational requirements of the code completely predictable. Consequently it was possible to manually examine the source code and extract very detailed information about the computational and memory bandwidth requirements of the application.

4.5 Analysis Results

Table 4-a shows the number of additions and multiplications required per site for each of the three operations that are used in the conjugate gradient application. Also

	Proportion of Conj Grad	FP Ops/ Mem Ops	Add Percentage	Multiply Percentage
Dirac	95.6%	6.81	55%	45%
Dot Product	1.8%	1	50%	50%
Add-Scale	2.6%	0.66	50%	50%

Table 4-b. Memory characteristics and arithmetic operation distribution for lattice QCD operations

shown is the total number of operations for the conjugate gradient application, which is performed using a number of applications of each of the three operators. The operations themselves are discussed in detail in section 3.1.3. The figure for additions includes subtraction operations. Subtractions are performed using addition units by inverting the sign bit of the second operand. Conjugate gradient also requires 2 divisions per iteration of the main loop. A normal problem size uses fifty million floating point operations per iteration so division constitutes only a tiny fraction of the arithmetic operations required for conjugate gradient. As a result I concentrated on the allocation of addition and multiplication resources.

It can be seen from Table 4-a that the Dirac operator is by far the most computationally intensive of the lattice QCD operations used to implement the conjugate gradient application, requiring nearly 55 times more operations per site compared with either the dot product or matrix-add scale operation.

Table 4-b shows what percentage of the conjugate gradient calculation each operation represents. The ratio of arithmetic operations to memory accesses for each of the lattice QCD operations is also shown, along with the percentage of arithmetic operations that are additions and multiplications. The Dirac operator is used twice per iteration of the main loop of conjugate gradient algorithm whilst the matrix add-scale is used three times and the dot product is used twice. Consequently the Dirac operator forms over 95% of the computational load of the conjugate gradient algorithm.

It was determined that the Dirac operator is potentially amenable to an FPGA implementation since it has a high ratio of calculation to memory operations. This indicates that the application is bound by computational performance and not memory bandwidth. Further analysis of the algorithm was required to determine what parallelism is available in the algorithm, and to determine how to exploit this parallelism.

Unfortunately the other operations are limited by memory bandwidth, making it difficult to exploit any available parallelism. This is not a significant barrier to performance since they form only a small part of the overall calculation load for the conjugate gradient application.

4.6 Parallelism in the Dirac operator

The memory access characteristics of the Dirac operator show the potential for extensive exploitation of parallelism within the Dirac operator, assuming that the operator has exploitable parallelism. The dependencies within the operator were analysed to find exploitable parallelism in the Dirac operator.

A dependency diagram of the components of the Dirac operator is shown in Figure 4-c. This diagram shows that the algorithm involves a series of steps and that some of these steps consist of a set of independent operations. See section 3.1.3 for an explanation of what each of the function types involves. The first stage is the *gamma-mul* pairs. There are eight *gamma-mul* pairs and each is independent of the other seven, making them parallelisable. The *gamma-mul* pairs are the most compute intensive part of the Dirac operator making it possible to dramatically improve performance through parallelisation of these *gamma-mul* pairs.

The second stage in the Dirac operator adds the results of all the eight *gamma-mul* components, which are small matrices of complex numbers, together into one matrix. Seven matrix additions are required to do this. This is a reduction operation so some, but not all, of these additions are independent. The reduction is performed in three steps; the eight results are added together to make four intermediate results, these four are added to make two and then the two are added to make one. The matrix adds

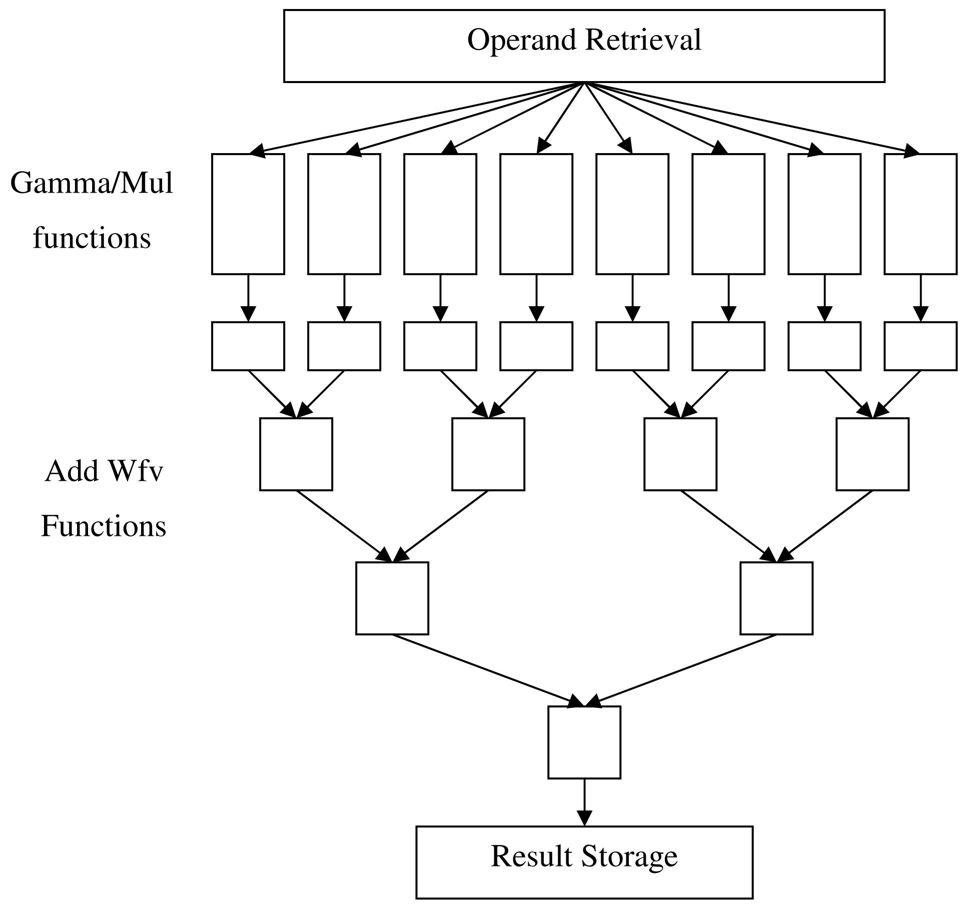


Figure 4-c. Diagram of the dependencies between operations in the Dirac operator

within each of these steps are independent, however the adds between steps are dependent. Thus parallelism is possible within the steps, but there is not as much potential as in the first stage.

This analysis has shown that the Dirac operator has low memory bandwidth requirements compared with its arithmetic processing requirements. Also internally the operator has a considerable amount of exploitable parallelism. These two characteristics indicate that the Dirac operator is compatible with the FPGA environment.

4.7 Parallelism in the Dot Product Operator

The dot-product operator is primarily memory bandwidth bound. Table 4-b shows that one memory operation is needed for each floating point operation in the dot-product operator. Consequently performance of the dot-product operator is memory bandwidth bound on most platforms. Since memory bandwidth is determined by the architecture of the FPGA board there is little scope to improve performance through efficient design on the FPGA.

The ADM-XRC II board used in this work provides 6 banks of 32-bit memory, which are clocked using the main FPGA clock. The theoretical maximum performance for the dot-product on this board is 6 floating point operations per cycle for single precision and 3 floating point operations per cycle for double precision. These figures are dependent on data layout in memory being optimal for the dot product operator. The Dirac operator is the dominant part of the conjugate gradient application however so data layout will be optimised for this part of the application. Consequently the performance of the dot product operator may be adversely affected.

4.8 Parallelism in the Matrix Add-Scale Operator

The matrix add-scale operator is even more memory bandwidth bound than the dot-product operator. This is because two separate operand matrices must be read and one of them must be updated with the result, making for a ratio of three memory operations to every two arithmetic operations.

The matrix add-scale operates by scaling the value for a point in one matrix by a fixed value. The result is added to the value for the same point in a second matrix. The result of this is then written over the previous value for the current point in the second matrix. The high memory bandwidth requirements for this operator mean that the theoretical maximum performance for the ADM-XRC-II FPGA board is 4 floating point operations per cycle for single precision and 2 operations per cycle for double precision. This assumes that the data layout in memory is optimised to the requirements of the matrix add-scale operator. However data layout will likely be optimised for the Dirac operator, which would adversely affect performance for the matrix add-scale operator.

4.9 Summary

The analysis presented in this chapter has shown that the most important part of lattice QCD calculations, the Dirac operator, is computationally bound and has low memory bandwidth requirements compared with computational requirements. The analysis has also shown that the Dirac operator has a substantial amount of exploitable fine-grain parallelism. These characteristics indicated that the Dirac operator can be efficiently implemented on an FPGA. The analysis also examined a full lattice QCD application called a conjugate gradient solver. This application uses the Dirac operator along with two other operators, the dot-product and matrix add-scale operators. These two operators are less well suited to an FPGA implementation since they are both memory bandwidth bound with comparatively low requirements for computational resources. However they only form a small part of the total load of the conjugate gradient algorithm. The result of this analysis was used to guide the implementation of the designs detailed in the next three chapters.

Chapter 5

Lattice QCD Using Logarithmic Arithmetic

The previous chapter presented a detailed analysis of the lattice QCD algorithms that are implemented in this work. The analysis showed that the performance critical Dirac operator has a lot of exploitable fine-grain parallelism, and that it has low memory bandwidth requirements compared with its computational requirements. Consequently an FPGA implementation of the Dirac operator has significant potential for good performance. This chapter begins by discussing the implementation, using logarithmic arithmetic, of the Dirac operator for FPGAs. Whilst the Dirac operator is the most significant part of lattice QCD simulations, a full application uses a number of other operations that can have a significant effect in performance. To evaluate the effect of these operations a full conjugate gradient solver application was implemented for FPGAs and this is described in this chapter.

5.1 Log arithmetic Dirac operator implementation

This section begins with analysis of the resource requirements of logarithmic arithmetic cores. The analysis is used to determine how many units can fit on the FPGA used in this work. This information is then used to determine how to allocate the available units within the final Dirac operator design, and this design is discussed in detail. The design discussion starts with a description of the design of the small matrix operators that the Dirac operator is built from. These operators are used to create an "application pipeline". This pipeline parallelises all memory operations with a calculation step and it parallelises the *gamma-mul* blocks with the matrix addition stages. These stages are discussed in detail in the previous chapter, Chapter 4.

Two appendices which describe two aspects of the logarithmic Dirac operator design process are attached to this thesis. Appendix A describes in detail how the arithmetic unit pipelines are used within the log arithmetic Dirac operator design. Appendix B describes the process used to improve the clock rates of the designs. Initial versions of the designs had quite poor clock rates which were improved through an iterative process of clock rate improvement. The appendices are included to illustrate two important aspects of the design process used in this work and are complementary to the design description contained in this chapter.

5.1.1 LNS arithmetic unit analysis

Table 2-a, Table 2-b and Table 2-c in Chapter 2 show the resource requirements for the logarithmic arithmetic units. Logarithmic multipliers and dividers are very small and have short pipeline latencies. Consequently the supply of multiplication resources was not a constraint for the designs. The logarithmic adders are much larger, and they have a significant requirement for block RAM. Logarithmic adders must be instantiated in pairs and a pair of adders requires 28 block RAMs. Thus no more than ten adder pipelines can fit on the FPGA used in this work, because the FPGA available has 144 block RAM elements.

The characteristics of the lattice QCD operators, shown in Table 4-a show that the operators have a reasonably balanced requirement for additions and multiplications, with the requirement for addition being slightly higher. This makes the availability of addition resources the critical performance constraint for the Dirac operator. The ratio of adds to multiplications for the core Dirac operator is 11 adds to 9 multiplications. Ten adders are available on-chip so no more than an average of nine multiplications per cycle can be performed, giving a possible peak throughput of nineteen operations per cycle for the core Dirac operator. The multipliers are small allowing use of a significant number to ensure that the adder pipelines are used efficiently within the designs. As discussed in Chapter 4 the other operators in the conjugate gradient algorithm are limited by memory bandwidth so availability of arithmetic units is not a limiting factor for these operators.

5.1.2 LNS arithmetic use in the Dirac operator

The eight g*amma-mul* operations are the dominant part of the Dirac operator. Multiplication of the 4×3 by the 3×3 complex number matrices (introduced in section 3.1.3) requires 264 floating point operations per matrix multiplication. The *gamma* preconditioning takes a further 12 operations. Internally the result of the *gamma* operation is an operand to the matrix multiplication.

The *gamma-mul* operations are independent, so they can be parallelised given sufficient resources. One adder can be dedicated to each *gamma-mul* operation with each adder performing 126 operations. This will only work if the *gamma-mul* block can be implemented efficiently using a single adder. The multipliers are small so several could be used with each adder pipe, even if they are underutilised, to ensure that the adders are used to the maximum extent in each *gamma-mul* block.

In this scheme, with a single adder dedicated to each of the 8 *gamma-mul* blocks 2 adders are not used by the *gamma-mul* blocks. These two adders are used to sum the results of the *gamma-mul* blocks in parallel with the *gamma-mul* blocks. Using two adders for the addition section works well; 192 additions and subtractions must be performed so each pipe performs 96 operations. Thus these adders have similar computational loads to those adders used for the *gamma-mul* blocks. In this scheme the computational loads on the adders are well balanced and there is less resource sharing within the design, which helps to maximise clock rate.

Operand data for the *gamma-mul* blocks is used several times after it has been read in from off-chip RAM. Streaming data directly from off-chip RAM to the arithmetic units is not practical since the data will need to be read multiple times, inflating memory bandwidth requirements. Thus for the designs operand data is stored in on-chip distributed RAM (all block RAM is used by the LNS adders), making it impossible to issue more than two operations per cycle that involve any single operand matrix. Thus dedicating an adder block to each *gamma-mul* block was determined to be the best strategy.

5.1.3 Gamma-Mul Block Design

This section describes the design of the hardware used to perform the *gamma-mul* block calculations. One adder and two multipliers are used to implement each block. There are eight blocks in total which are all structurally identical. There are some small differences between the blocks, for some blocks numbers are added but in other blocks the equivalent numbers are subtracted. However the same design can be used for all eight blocks with some small adjustments. Figure 5-b shows the structure of a log arithmetic *gamma-mul* block.

The *gamma-mul* blocks have two parts. The first part is the *gamma* calculation. Each gamma operation consists of 12 additions or subtractions, and the operands for these additions and subtractions are elements from a single complex number matrix. The result of each of these additions is stored into two locations in the result matrix. The operands for the gamma operator are stored in 12-element distributed RAMs (one each for the real and imaginary halves of the matrix). Data is then read from these RAMs, processed by the adder pipeline and stored in another pair of distributed RAMs.

Each *gamma* operator is paired with a matrix multiply operator. The results from the *gamma* operator are the operands for the matrix multiply operator. Thus the matrix multiply operators read their operands from the distributed RAMs (see section 2.1) used to store the results of the *gamma* operator, once the *gamma* operator has completed.

Figure 5-a shows the source code for a complex number matrix multiply routine which forms part of a *gamma-mul* block. Most of computational load in the Dirac operator is in these matrix multiply routines. Each component of each point in the result matrix requires 6 multiplications and at least 5 additions. The value for each component is calculated by summing the results of 6 multiplications in a reduction type operation.

```
for (d = 0; d < 4; d++)
  for (c = 0; c < 3; c++){
    for ( b = 0; b < 3; b++){
      b[d][c].r += g[b][c].r*a[d][b].r + g[b][c].i*a[d][b].i
      b[d][c].i += g[b][c].r*a[d][b].i - g[b][c].i*a[d][b].r
```

Figure 5-a. Source code for a complex number matrix multiply routine. This code, combined with a gamma operation makes up a *gamma-mul* block

The summation has three levels of dependency for each component:

- The six multiplications are independent
- The results of the multiplications are added in to 3 partial results
- The partial results can then be summed in two dependent stages

When used with a single adder pipeline, the code in Figure 5-a causes dependent operations to be scheduled sequentially. No-ops must be issued to the adder pipeline for the code to execute correctly. Re-ordering the loops so that the *b* loop is the outermost loop reduces the impact of the data dependencies. Instead of performing all the operations for each point before moving on to the next, an operation is performed for each point and then a second operation is performed for each point and so on. This replaces the no-ops with useful operations, improving performance significantly.

Two multipliers are used to process the multiplications. The results of the multiplications are sent directly into the adder to perform the first stage addition for the point, allowing the multiplications and first stage additions to be streamed through the two multipliers and the adder. This creates three partial results for each component which are stored in temporary storage. Two 36 element dual port distributed RAMs (see section 2.1) are used to store the partial results for the real and imaginary halves of the matrix.

Once all of the multiply results have issued to the adder, the adder pipeline is available to start adding the partial results stored in the two 36 element distributed RAMs. The RAMs are dual ported so the partial results retrieved and then issued to

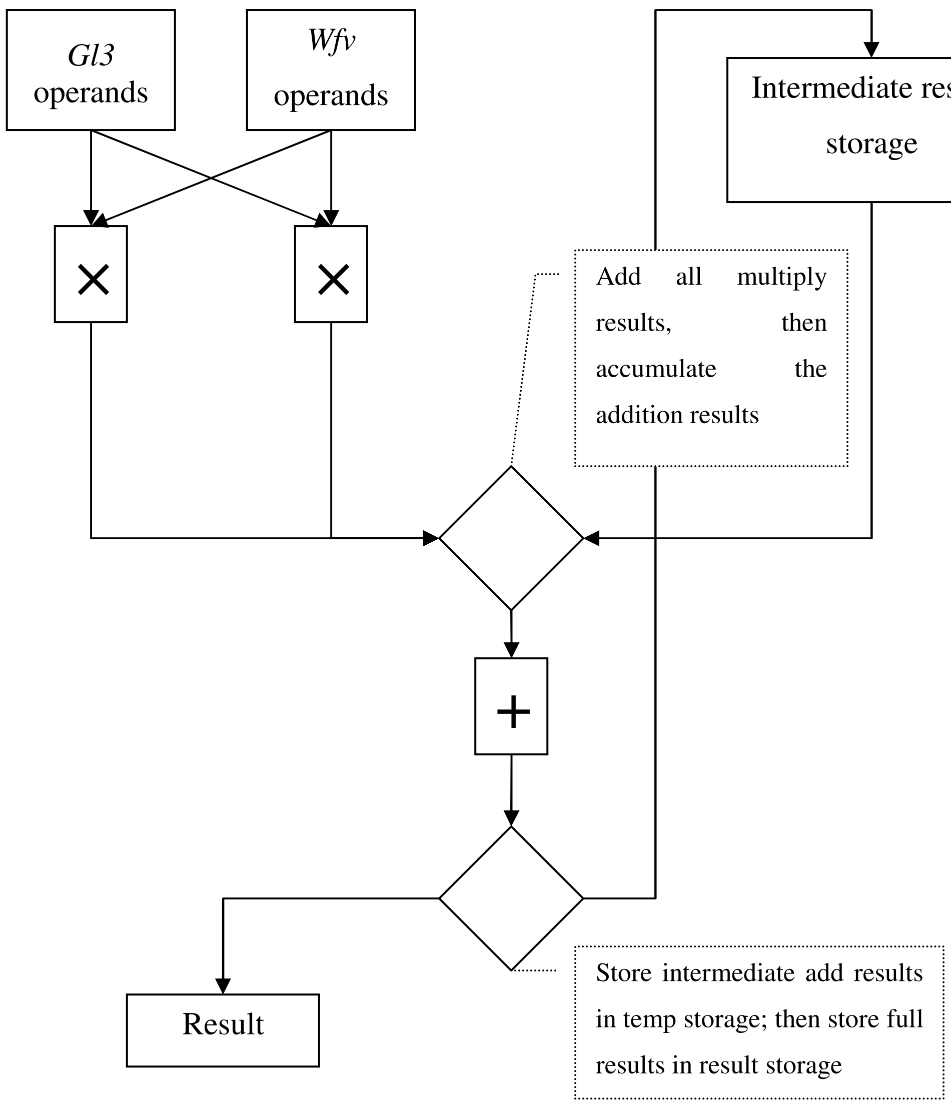

Gl3 operands
Wfv operands
Intermediate result storage
×
×
Add all multiply results, then accumulate the addition results
+
Result
Store intermediate add results in temp storage; then store full results in result storage

Figure 5-b. Logarithmic arithmetic *gamma-mul* block

the adder, at the same time as the last of the partial results are stored from the adder output into the distributed RAMs. The architecture of the logarithmic *gamma-mul* block is shown in Figure 5-b.

The addition of the partial products is performed using an accumulate type operation. There are 24 sets of 3 numbers to be added together, and since the pipeline latency is 9 cycles, these sets of partial results can be accumulated into the result matrix without any stalls on the adder pipeline.

```
macro proc IssueSD(ra, rb, ia, ib){
 par{
       IssueSDX=0;
       doIssueSDX = 1;
 }

 do{
       par{
             if(IssueSDX == 11){
                   doIssueSDX = 0;
             } else {
                   IssueSDX++;
             }
             la9(ra[IssueSDX], rb[IssueSDX]);
             la10(ia[IssueSDX], ib[IssueSDX]);
       }
 }while(doIssueSDX);
}
macro proc RetrieveSD(rr, ir){
 par{
       retrieveSDX = 0;
       doRetrieveSDX = 1;
 }

 do{
       par{
             if(retrieveSDX == 11){
                   doRetrieveSDX = 0;
             } else {
                   retrieveSDX++;
             }
             rr[retrieveSDX] = las_res9;
             ir[retrieveSDX] = las_res10;
       }
 }while(doRetrieveSDX);
}
```

Figure 5-c. Operand issue and result retrieval function used to implement matrix addition in the Dirac operator pipeline.

5.1.4 Matrix Addition, Matrix Subtraction and Matrix Scale Designs

The matrix addition operations are much simpler than the *gamma-mul* sections. Addition of two complex number matrices requires that the equivalent values for each point in the two operand matrices be added together. The hardware implementation of this operation is quite straightforward. As previously discussed in section 5.1.2 two adder pipelines are used for this part of the Dirac operator. The additions are streamed through the two available adder pipelines, by issuing the operands to the pipelines and then retrieving the results when they become available. The results are then stored in temporary storage. The matrix subtraction and matrix scale operators are implemented using the same approach.

Initial implementations of these operators used a single operator block which issued operands to the arithmetic pipeline and then retrieved the results and stored them in temporary storage. The design was later changed to separate the issuing of operands and the retrieval of results into two separate operations. Performance for the Dirac operator pipeline (discussed in the next section) was limited by the addition of the results of the *gamma-mul* stage of the algorithm. In the Dirac operator pipeline the addition operator is used several times in succession, and using a single operator block meant that the adder pipelines were being flushed unnecessarily. Separation of operand issue and result retrieval, as shown in Figure 5-a, reduced flushing of the adder pipeline significantly, and improved performance for the Dirac pipeline.

5.1.5 Dirac Operator Pipeline

Lattice QCD represents a very small section of 4-dimensional space-time using a matrix. As part of a lattice QCD calculation the Dirac operator updates each point in the matrix once to complete what is termed a sweep of the matrix. Each point update within a sweep is independent of any other update in that sweep. The application of the Dirac operator to a point in the matrix depends only on the results from the previous sweep.

Consequently it is possible to perform the *gamma-mul* operations for one site at the same time as the addition operations for another site. A pipeline is used to exploit this, which performs the addition stage for site *n* at the same time as the *gamma-mul* section for *n+1*, creating a two stage pipeline which parallelises the *gamma-mul* and addition stages.

Operand Retrieval and Result Storage

All the operand and result data for the Dirac operator design is stored in off-chip memory. When needed the data is read into on-chip temporary storage. Small distributed RAMs are used to implement this storage. Reading in this data takes time however and this can adversely affect performance of the entire design, unless it is parallelised with other parts of the operators.

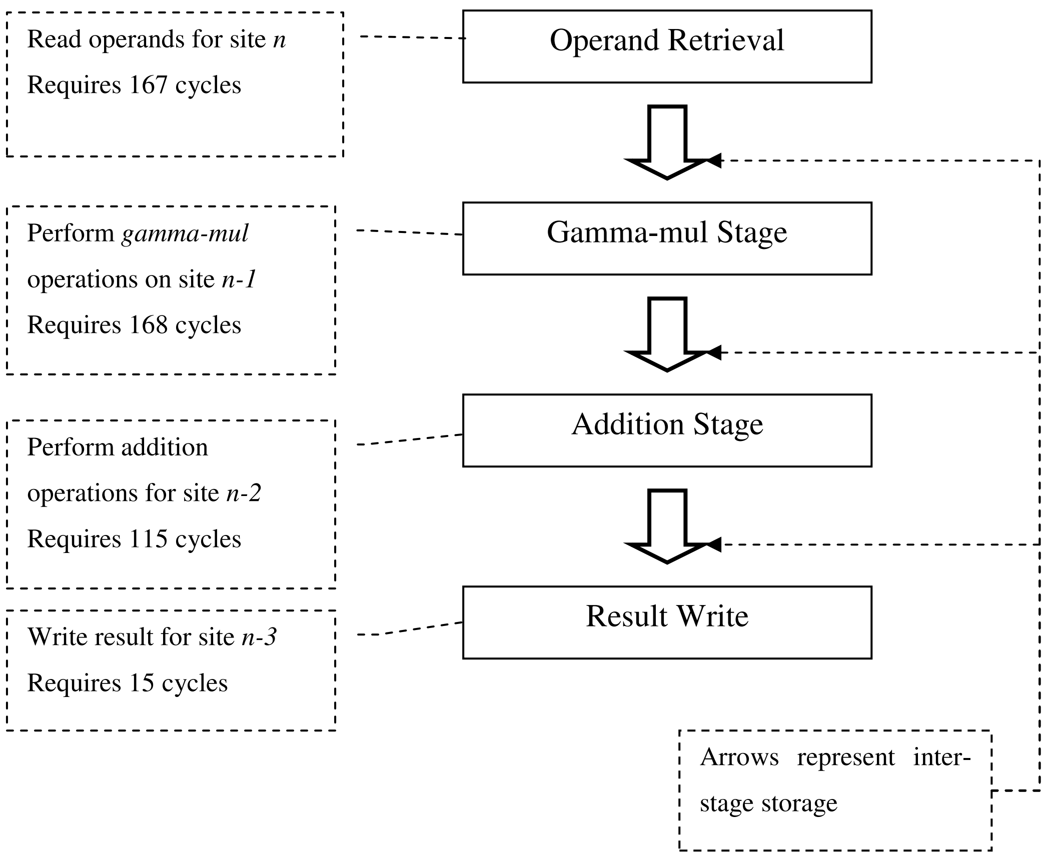

Read operands for site *n*
Requires 167 cycles
Operand Retrieval
Perform *gamma-mul* operations on site *n-1*
Requires 168 cycles
Gamma-mul Stage
Perform addition operations for site *n-2*
Requires 115 cycles
Addition Stage
Write result for site *n-3*
Requires 15 cycles
Result Write
Arrows represent inter-stage storage

Figure 5-d. Diagram of the logarithmic arithmetic Dirac operator pipeline, including cycle count information for each stage

Two extra stages were added to the Dirac operator pipeline. These two stages read in operand data and write out result data in parallel with the two computation stages. This created an application pipeline with four stages which processes 4 sites at once. The stages, and the number of cycles each stage takes to complete, are shown in Figure 5-d

To apply the Dirac operator to a site requires 8 *gl3* type matrices and 9 *wfv* type matrices to be retrieved from memory (the *wfv* and *gl3* matrices are described in section 3.1). A single *wfv* matrix, the result, must be stored to memory also. The *gl3* matrices are retrieved from a large *g* type matrix which, I will call *H*, whilst the *wfv* matrices are retrieved from a large *y* type matrix which will be called *A*. The result *wfv* matrix is stored into another large *y* type matrix, which will be called *B*. The *g*

	Number of Matrices	**Number of 32-bit data words**
Reads from *H*	8	144
Reads from *A*	9	216
Writes to *B*	1	24
Total Ops	18	384

Table 5-a. Memory bandwidth requirements per site update for Dirac operator

and *y* type matrices are described in section 3.1.3.

Table 5-a shows the number of reads and write for each of the large operand matrices, along with the total number of operations required per site. The Alpha-Data ADM-XRC II development board used in this work can perform up to six 32-bit memory operations per cycle. 384 operations must be performed per site, at a maximum rate of 6 operations per cycle so, including the 4 cycle latency of the SRAMs, a minimum of 68 cycles are required to perform all the memory operations for a single site.

Achieving this would require the data for each of the three matrices to be stored across all six banks. This is a complex layout and would be difficult to manage; it is better if the data for each large matrix is stored in as few banks as possible. The simplest approach, which is used in this design, is to store each matrix, (*H, A* and *B*) in a dedicated pair of memory banks, putting the real component of each number in one bank and the imaginary component in the other. This data layout requires a minimum of 112 cycles to read in all the operand data.

The cycle count for the actual implementation is higher at 167 cycles, since the operand retrieval stage must wait until the first calculations stage has read all of the data for the previous site, before operand retrieval can commence. However the operand retrieval requires fewer cycles than the first calculation stage, so this does not restrict performance.

The Gamma-mul Stage

The first calculation stage consists of eight *gamma-mul* blocks. All eight *gamma-mul* operators are independent and are run in parallel. Each of the blocks reads its operands from small distributed RAMs and writes its result to another set of distributed RAMs. The operands were written to the input RAMs by the operand retrieval stage on the previous iteration of the application pipeline. The results are read from the output RAMs by the addition stage on the next iteration of the pipeline. The *gamma-mul* blocks are the most computationally intensive part of the Dirac operator and parallelising them improves performance dramatically.

The Addition Stage

The addition stage has a dedicated pair of adders. It is constructed using the matrix add, subtract and scale blocks described in section 5.1.4. This stage accumulates the eight results produced by the *gamma-mul* stage into a single result matrix. This result is then subtracted from another matrix and the result of that is scaled by a value.

Early implementations of the design flushed the two adder pipelines used in this stage between each use of the matrix add operator. Each flush took 10 cycles which caused the *add-wfv* stage to take longer to complete than the *gamma-mul* stage. The flushes were eliminated by separating the issue and retrieval into two separate operators, as previously discussed in section 5.1.4. This allowed all the arithmetic operations to be issued to the adder pipelines without any pipeline flushes. The issue operators are called in parallel with the retrieval operators, with the retrieval operators delayed so that they start retrieving results from the adder pipelines as soon as they are available.

The results of all the additions are stored in one of seven pairs of distributed RAM arrays. Some of these arrays are dual port RAMs because their contents need to be read by the issue operators whilst results are still being written to them by the result retrieval operators. The matrix scale and matrix subtract operators are combined to reduce the number of cycles required for these two operations. The results of the matrix scale operation are fed directly from the multiplier pipeline outputs into the adder pipelines, which eliminates any need for temporary storage.

Result Write Stage

The result write stage is very simple, it writes out the result from the previous iteration of the pipeline, stored in a small distributed RAM, into off-chip memory. A total of 24 32-bit words must be written into two memory banks, and the memory banks have three cycle latency for write operations so a total of 15 cycles are required for this stage. The result must be written to memory before it is overwritten with the next result by the previous pipeline stage, but this is easily achieved since there is a period of the order of one hundred cycles before the previous stage starts writing to the distributed RAM.

5.2 Log arithmetic conjugate gradient solver

5.2.1 LNS Matrix Add-Scale and Dot Product operators

The dot-product is the sum of the square of all the real and imaginary components of all the values in one of the large matrices that are part of a lattice QCD dataset. The dot-product is used to determine if the conjugate gradient solver has reached a solution. During each iteration, the solver calculates the dot product and compares the result to the dot product result from the last iteration of the solver. A solution has been reached when the difference between the two dot product results drops below a predetermined threshold. The dot product operation is shown in equation (15) below.

$$d = \sum_{i=m}^{n} (X_i \times Y_i) \tag{15}$$

The dot product operator used in the version of the conjugate gradient solver that was implemented uses the same matrix for both operand matrices, which reduces memory bandwidth requirements. The operand matrices are the large *Y* type matrices from the calculation dataset (described in section 3.1), which are actually matrices of small complex number matrices. However for the purposes of the dot product and matrix add scale operators they can considered as vectors of complex numbers.

$$y_i = (k \times x_i) + y_i \tag{16}$$

$$y_i = (k \times y_i) + x_i \tag{17}$$

The scale-add operation has two forms which are shown in (16) and (17). In both forms every value from one matrix is multiplied by a constant value, and the result is added to the equivalent point in another matrix. The difference between the two forms is in where the result is stored. Any implementation of this algorithm will have high memory bandwidth requirements compared with computational requirements. Three memory operations and two arithmetic operations are required to calculate the result for each location in the result matrix, which makes performance for this operation memory bandwidth bound for most platforms.

5.2.2 Memory Layout

The dot product and matrix add-scale operators are both memory bandwidth bound for the FPGA development boards used for this work. This is the case for most computing platforms used for lattice QCD. For single precision arithmetic the Alpha-Data ADM-XRC II board used here can sustain a maximum of 6 floating point operations per cycle for the dot-product and a maximum of 4 operations per cycle for the matrix add-scale. To put the lack of memory bandwidth in context, the FPGA used here has sufficient arithmetic resources to sustain 20 operations per cycle for either of these operators.

The dataset for the Dirac operator is a subset of the conjugate gradient solver dataset. The difference between the two is that the conjugate gradient solver has five *Y* type matrices, where the Dirac operator has only two. See section 3.1 for details on the large matrices that form the Dirac operator dataset. Maximum performance for the dot-product and matrix-scale operators can only be achieved using a very complex memory layout pattern that spreads all the *Y* type matrices across all six memory banks. Unfortunately such a memory layout would make memory access logic very complex for the performance critical Dirac operator.

The data layout used for the Dirac operator stores each *Y* type matrix in a pair of

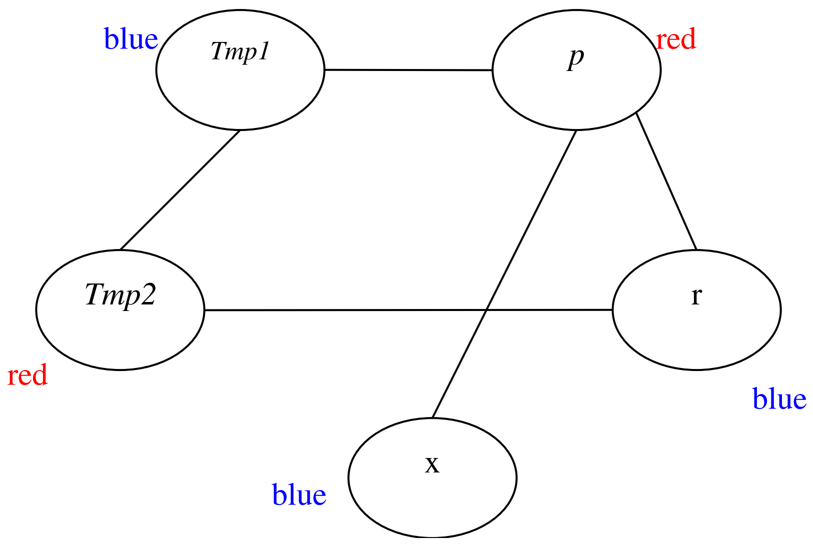


Figure 5-e. Usage of the *Y* type matrices used by the conjugate gradient solver shown as a graph colouring problem

SRAM banks. The real components of each number are stored in one bank and the imaginary components are stored in the other bank. This limits memory bandwidth for the dot product and matrix add-scale operators. However the Dirac operator was already complete before work started on the full conjugate gradient application. Changing the data layout would have required substantial re-engineering of the completed Dirac operator and it was decided that the performance improvement from doing this would not be significant enough to justify the work involved.

The conjugate gradient operator dataset includes five *Y* type matrices. Each of these is stored in a pair of SRAM banks. Two pairs of SRAM banks are dedicated to storing the *Y* type matrices. If possible no two matrices used by one call of any of the operators should be stored in the same bank. If two matrices are stored in the same bank then memory bandwidth would be substantially decreased for that operator.

Figure 5-e describes the relationships between the *Y*-type matrices used in the conjugate gradient application as a graph colouring problem. When two matrices are used by the same call of an operator they are connected by a line. No two matrices that are connected by a line can have the same colour. The graph can be coloured with two colours so two pairs of banks are required to store all the matrices. This means that any unconnected matrices can be stored in the same SRAM bank

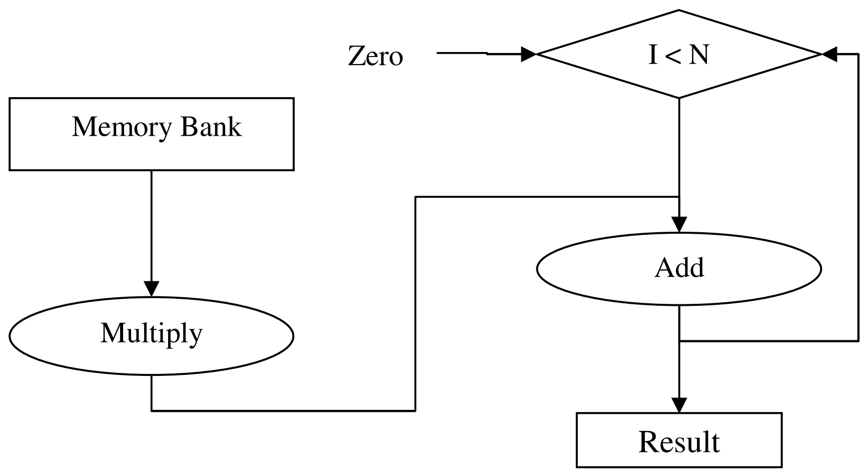


Figure 5-f. Dot product architecture

Therefore *p* and *temp2* are stored in SRAM banks 4 and 5 and the other three matrices are stored in banks 2 and 3.

5.2.3 Log arithmetic dot product operator

Figure 5-f shows the architecture of the implementation of the dot-product operation. The real and imaginary components of each number in the matrices are stored in separate banks. This allows two of the architectures to be run in parallel and the results can then be added at the very end. One adder and one multiplier are dedicated to each instance of the dot-product operator.

The architecture streams data in from memory and issues it directly to the multiplier to square it. The result of the multiply is then added to one of *n* running totals, where *n* is the latency of the adder (9 cycles for the log arithmetic adder). This is achieved by issuing the multiply result to the adder along with a zero for the first *n* multiply results. After *n* results have been issued to the adder the result of the first addition is available from the adder's output. This result is re-issued to the adder along with the current multiply result. This is continued until all the numbers in the matrix have been squared and added to one of the *n* running totals.

Two of these architectures are run in parallel so two sets of *n* running totals are left once the matrix has been processed. These are added using the following method:

- On the cycle after the last multiply result is issued to the adder, the first of each set of running totals from each instance of the architecture is available from each adder.

- These results added by issuing them to one of the adders. All elements of the two sets of running totals are added in this way.
- Now there are *n* running totals to be added. There are 9 running totals for the log arithmetic implementation so this is done in 4 stages using a reduction type operation. First 8 results are added to leave 5 totals. Then 4 are added to leave 3, then 2 are added to leave 2 and finally the last 2 are added to give the final result.
- This requires about 50 cycles to complete.

The architecture is very efficient, although it would complete 9 cycles faster if the latency of the adder pipeline latency was 8 cycles instead of 9 because the last add would be eliminated. The matrices that form the input to the operator have tens of thousands of elements so that the final summation forms only a very small portion of the time spent in the operator.

5.2.4 Log arithmetic matrix add-scale operator

There are two different operations that are quite similar so they were treated them as one operation. The operations are described by equations (16) and (17) in section 5.2.1. It was determined that if three parameters were passed to the operator then a single operator could perform both operations easily. The parameters are: the address of the matrix to be multiplied, the address of the matrix to be added and the address of the matrix where the result is to be stored. This unified operator is used by simply passing the appropriate addresses for the two operands and the result to the one operator. For a matrix scale-add operation the multiply and result addresses are the same whilst for a matrix add-scale the add and result addresses are the same.

The *Y* type matrices are stored in the off-chip memory banks, as described in 5.2.2. Unfortunately data must be both read from and written to one of the matrices which restricts throughput on the arithmetic units. It was decided to stream data from the off-chip memory, reading operand data every second cycle. The results were then was then stored in these spare cycles. The operands arrive at the relevant arithmetic units in the correct cycle and are used straight away.

This approach processes the entire matrix without stopping, thus maximising performance. The design uses a single adder and a single multiplier. The calculations of the real and imaginary components of each number for the matrix are alternated onto these arithmetic units. The alternative approach was to buffer the results and write them out to off-chip memory in batches, however no block RAMs were available to buffer the data so this approach was not viable.

5.3 Summary

This chapter has described the design and implementation of an FPGA based Dirac operator using logarithmic arithmetic. This is the most important part of most lattice QCD calculations, and if a platform has good performance for this operator than it is likely to have good performance for complete lattice QCD applications. However a full application includes a number of other operations which can have a significant effect on performance. To evaluate the effect of these operations, a full lattice QCD conjugate gradient solver application was implemented using logarithmic arithmetic. The design of this application and the operators it requires are described in this chapter.

As described in section 3.2.2 either single or double precision arithmetic can be used for lattice QCD. However double precision is preferred since there are some uncertainties over the stability of single precision for lattice QCD simulations that are run at the most physically interesting parameters. During this project double precision IEEE arithmetic cores became available. The next chapter details the design and implementation of the Dirac operator and the conjugate gradient solver application using these double precision cores.

Chapter 6

Lattice QCD Using IEEE Double Precision Arithmetic

This chapter describes implementations of the lattice QCD Dirac operator and conjugate gradient application that use IEEE compliant double precision arithmetic cores. The use of double precision arithmetic on FPGAs presents a significantly different challenge to the use of logarithmic arithmetic, as used in the designs described in the previous chapter, Chapter 5. This chapter starts with a discussion of the extra performance constraints that double precision designs are subject to, compared with logarithmic arithmetic designs.

The chapter continues with an analysis of the resource requirements of the double precision arithmetic cores that were used in this work. This analysis is used along with the application analysis from Chapter 4 to determine what mix of arithmetic units would best suit the lattice QCD algorithms. The design of a Dirac operator and conjugate gradient solver application using these cores is then described. Whilst this design has some similarities to the logarithmic designs, it is much more complex and is more heavily constrained than the logarithmic designs.

Double precision arithmetic is the preferred arithmetic system for lattice QCD simulations, as detailed in 3.2.2. Consequently the designs described in this chapter are more significant than the logarithmic arithmetic based designs described in the previous chapter. Furthermore since double precision arithmetic is the most commonly used non-integer arithmetic system for high performance computing, applications the results of the designs in this chapter offer significant insights into the suitability of FPGAs for general scientific computing.

6.1 Design Constraints for Double Precision Arithmetic

The double precision implementations are designed to maximise parallelism whilst maintaining a high clock rate in order to maximise performance. Performance for the log arithmetic designs is limited by the number of available adders. The limited number of adders keeps the demand for memory bandwidth low. Meanwhile the log arithmetic multipliers are very small compared with the adders so it was not important that the multipliers be efficiently used in the designs. The logarithmic designs take advantage of the small multiplier size, by using more multipliers than strictly required by the balance of arithmetic operations in the Dirac operator, thus ensuring maximum utilisation of the adders. Essentially the target when designing with the logarithmic cores was to make best use of the adders.

The resource requirements of the double precision arithmetic units are much more balanced than for the log arithmetic units. The double precision adders and multipliers require similar amounts of FPGA slices and the multiplier also requires nine hardware multipliers, see Table 2-d and Table 2-e for details. As a result, high performance for the double precision designs could only be obtained through efficient use of both types of unit. Also, since double precision floating point data words are twice the width of single precision equivalent log arithmetic data words, memory bandwidth and data storage requirements are doubled compared with the log arithmetic implementations. This caused on-chip data storage and retrieval of data from off-chip memory to become critical performance constraints for the double precision applications. Therefore the critical design constraints for the double precision designs were:

- Limited availability of both multipliers and adders
- High memory bandwidth requirements

- Layout of memory in off-chip RAMs
- Shortage of resources on the FPGA

6.2 Double precision IEEE arithmetic unit analysis

The double precision designs used the Moloney cores [Moloney '04]. The characteristics of the adder and multiplier cores are shown in Table 2-d and Table 2-e. The resource requirements of the divider are not considered in this analysis since division is a necessary but insignificant part of the lattice QCD algorithms. These tables show that the resource requirements of the double precision cores are quite different to the resource requirements of the log arithmetic cores. The double precision multiplier and adder require a similar number of slices each (about 900), and the multiplier also requires 9 on-chip hardware multipliers. Finally the pipeline latencies are similar, at 6 cycles for the adder and 7 cycles for the multiplier.

Consequently the best mix of arithmetic units is one which reflects the computational mix of the algorithm, shown in Table 4-a. This is quite different to the log arithmetic implementations where the best mix is to have as many adders as possible and then sufficient multipliers to ensure that the adders are used efficiently. The Dirac operator is 55% addition and 45% multiplication. The dot product and matrix add-scale operators have an equal requirement for both multiplication and addition but are limited by memory bandwidth and not arithmetic resources.

To determine the optimal number and mix of arithmetic units, a number of factors were considered including the requirement for FPGA resources for the application control logic, and also the requirement for division in the conjugate gradient application. The resource requirements of double precision dividers are substantial and are shown in Table 2-f. The Moloney core used here is small for a double precision divider but still requires nearly 1800 slices, which is 5% of the available slices on the FPGA used in this work. Only one divider is required by the conjugate gradient application but it still requires a substantial amount of resources.

Many of the issues involved in exploiting fine-grain parallelism in an FPGA are already discussed in section 4.3. All data for lattice QCD applications are small complex number matrices. These matrices are stored on-chip using single or dual ported RAM. These RAMs limit exploitable parallelism within operations on these small matrices since no more than two operands can be retrieved from any matrix on a given cycle. Parallelism can be more effectively exploited by performing multiple matrix operations in parallel. This approach was successfully employed for the logarithmic implementations and it was successfully used for the double precision implementations as well.

The Dirac operator consumes 9 additions for every 8 multiplications (see Chapter 4) so the ideal balance of arithmetic units is in this ratio. Using 8 multipliers and 10 adders for the design gives a balance of units broadly in line with this, using 51% of the available slices. This leaves sufficient chip resources for application control logic and so is the balance chosen for the design.

If more arithmetic units were included in the design then there would be very little space for the control logic for the application. For example if 10 adders and 16 multipliers, the same balance as the log arithmetic Dirac operator, were instantiated then over 67% of the FPGA resources would be taken. This would leave little usable space for application logic. Having, say, 12 adders and 10 multipliers would give little advantage either since it would be difficult to use the extra units effectively across the entire application.

6.3 Double precision Dirac operator implementation

There are some similarities between the log arithmetic and double precision implementations. Parts of the log arithmetic implementations were used as a base for the equivalent parts of the double precision implementations. Many of the simpler components were converted with reasonable ease. The matrix addition operations are similar for both implementations and were converted by adjusting for the size of the floating point data words and for the different latencies of the arithmetic units.

The more complex and performance critical components of the double precision Dirac operator are very significantly different. For example the *gamma-mul* blocks needed to be completely redesigned to allow for the new mix of arithmetic units.

Also the off-chip memory control needed to be completely re-built to meet the memory bandwidth requirements of the double precision application. Finally a shortage of FPGA resources was a very significant problem for the double precision implementations; the size of the arithmetic units combined with the size of the double precision data meant that significantly more resources were required for the double precision implementations.

6.3.1 Use of On-Chip Block RAM memory

In the log arithmetic implementations nearly all the on-chip block RAMs are occupied by the adder pipelines so all on-chip storage was implemented using registers or distributed RAM. However for the double precision implementations all the block RAMs are available since none are used by the arithmetic units. The use of double precision arithmetic doubles on-chip storage requirements compared with single precision arithmetic. Consequently considerable effort was spent on ensuring efficient use of the on-chip block RAMs.

Access to block RAM is through a one cycle pipeline. In order to read data from the block RAM the address in the block RAM is presented to the address port of the RAM. The block RAM data outputs have attached registers and the result of the read is written into this register on the next clock edge. In order to maintain simplicity Handel-C, by default, clocks block RAMs at twice the system clock rate. This allows block RAMs used in a Handel-C design to be accessed in a single cycle. Unfortunately running the block RAMs at twice the system clock tends to severely limit clock rate for Handel-C designs that use block RAM.

To address this problem Celoxica, the producers of the Handel-C design tools, provide a compiler transformation (first introduced in version 3.0 of their tools) that can automatically pipeline block RAM use in a design. The transformation is applied if all data read from a block RAM is written into a single register, which cannot be written to from anywhere else in the design. The transformation implements the register using the registers attached to the block RAM outputs. Registering the output of the block RAMs for the double precision Dirac operator added complexity to an already complex design. However without this the clock rate of the design would have been severely limited.

6.3.2 The gamma-mul pipeline stage

The analysis of the double precision arithmetic units in section 6.2 showed that using 10 adders and 8 multipliers best reflected the computational needs of the algorithm. This was sufficient to implement an application pipeline similar to that used for the logarithmic arithmetic Dirac operator described in Chapter 5. The design of this double precision Dirac operator is described in this section.

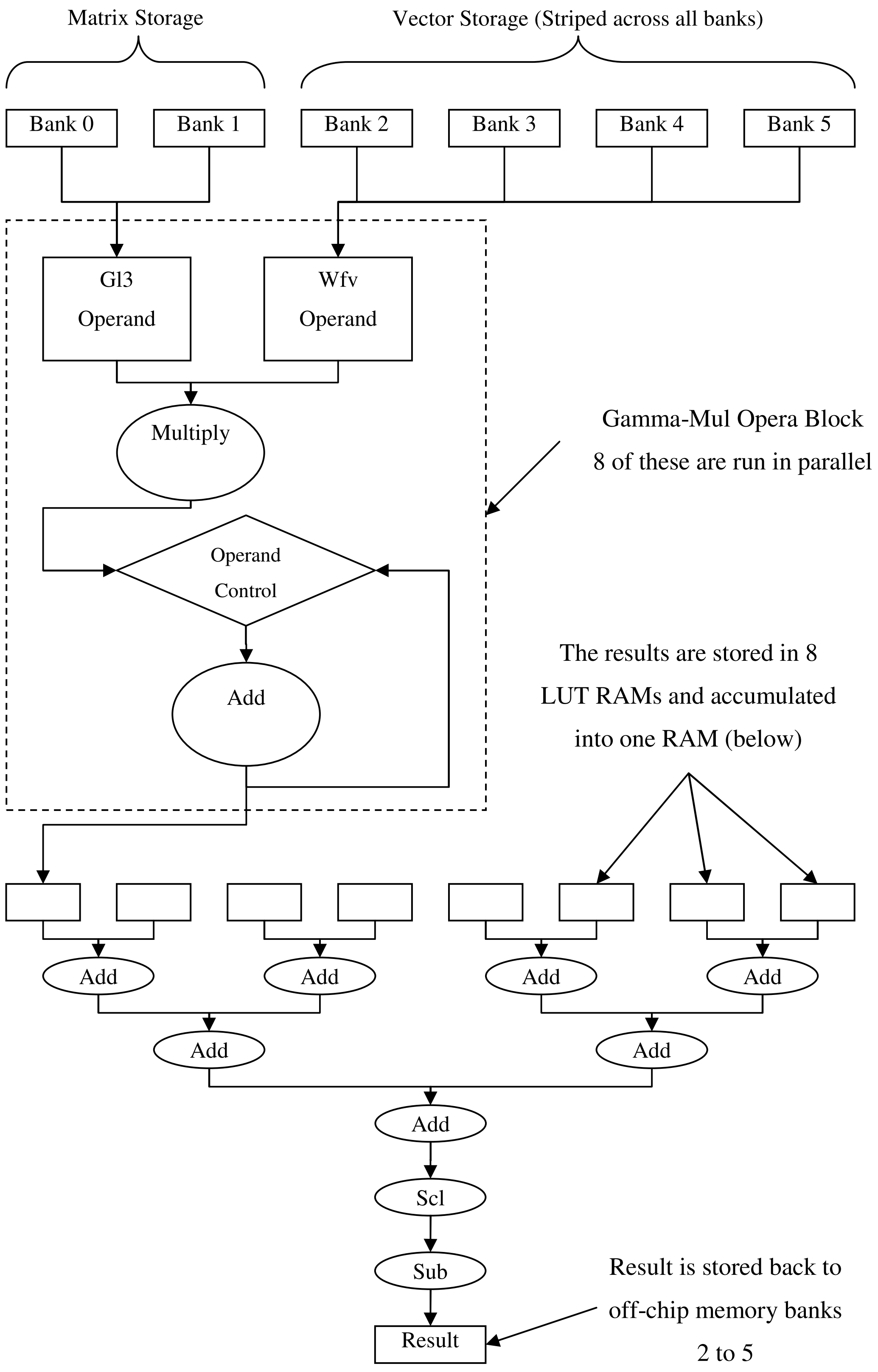
Gamma Operations

The gamma operations consist of 12 additions or subtractions whose result is a *wfv* matrix. The result of each addition is stored in two locations in the result *wfv* matrix. The *wfv* result matrix is stored in a dual ported on-chip RAM so these result stores would occupy both ports of the RAM if performed on a single cycle, preventing overlapping of the matrix multiply and gamma parts of the *gamma-mul* block.

This problem existed for the log arithmetic implementation also where it was solved using a multidimensional array of registers to store the gamma result. A register array of size *n* can support up to *n* parallel writes but requires a lot of FPGA resources. Every register in the array is connected to every point in the design from which the array is accessed. This creates a massive amount of multiplexing logic, which can be reduced to more manageable levels by using multidimensional arrays. The multiplexing logic for four 3-element arrays is much smaller than for a single 12 element array but it is still substantial. This was not a significant problem since was a lot of was plenty of spare resources available on the FPGA in the logarithmic designs.

A multi-dimensional register array could not be used for the double precision Dirac operator because FPGA resources were at a premium. Analysis of the access patterns for the *gamma* and *mul* sections of each of the eight *gamma-mul* blocks showed that the problem only occurred for half the *gamma* operations. By delaying one of the two writes to the gamma result RAMs by one cycle for the problem *gamma* operations, block RAMs could be used to store the *gamma* results with no loss of performance.

Figure 6-a. The architecture of the double precision Dirac operator. A diagram of the design of the *gamma-mul* blocks is inset.


Matrix Storage
Vector Storage (Striped across all banks)
Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Gl3 Operand
Wfv Operand
Multiply
Operand Control
Add
Gamma-Mul Opera Block
8 of these are run in parallel
The results are stored in 8 LUT RAMs and accumulated into one RAM (below)
Add
Add
Add
Add
Add
Add
Add
Scl
Sub
Result
Result is stored back to off-chip memory banks 2 to 5

Multiply operations

Since only 8 multipliers are available for the double precision Dirac operator the *gamma-mul* blocks had to be completely redesigned. The design of the *gamma* operations is discussed in the previous section. The *gamma* operations are combined with a complex number matrix multiply operation to give what are called the *gamma-mul* blocks. Using the two operations separately would incur a delay. Combining them eliminates this delay and improves performance.

Figure 6-a shows the structure of the double precision Dirac operator and inset is the structure of the *gamma-mul* block implementation. Each *gamma-mul* block uses 1 adder and 1 multiplier to multiply a 3x3 *gl3* matrix by a 4x3 *wfv* matrix to produce a *wfv* result matrix. The implementation uses the multiplier to produce 6 partial products for each component (real and imaginary) of each number in the result *wfv* matrix. The partial products must then be summed, giving the result for each component of each number. The issue order of the multiply operations is set so that the multiply results can be efficiently fed into the adder with minimal requirement for extra on-chip storage. The first two multiply results are partial products of site [0,0], the next two are of site [0,1] and so on up to site [3,2]. This sequence is repeated three times.

Since only one multiply result is produced per cycle, the multiply results can only be added on every second cycle. To handle this, the first result from the multiplier is registered and then issued to the adder in the next cycle; along with the current multiply result. These additions are referred to here as the stage-one additions. This process is continued until all the stage-one additions have been issued to the adder pipeline. At this stage two sets of three matrices remain; one set for the *r*-components of the result and the other set is for the *i*-components of the result. The three matrices in each of these sets must then be added together to get the real and imaginary components of the result matrix.

The summation of these result sets is overlapped with the stage one additions. As soon as the first stage one addition result is produced by the adder pipeline, it is accumulated to the correct location in the result matrix. The contents of the result matrix are set to zero beforehand. The free slots on the adder are used to perform these additions. Once the addition of the multiplication results has been completed then the accumulation additions are issued on every cycle thus making better use of the adder pipeline.

This operation is complex but it extracts excellent performance from both the adder and multiplier and requires minimal temporary storage for any intermediate results. Elimination of the temporary storage was important since resources are at a premium in this design. This scheme heavily utilises both the adder and multiplier pipelines and so returns excellent performance. Performance is further improved by the low latency of both the adder and multiplier. Low latencies reduce the pipeline fill time which improves performance.

6.3.3 Addition stage

Addition operations

The *add/subtract wfv* blocks are implemented by streaming data from the result RAMs of the *gamma-mul* blocks into two adder pipes; one pipe handles the real components and the other pipe handles the imaginary components of each number. This is the same approach as used in the log arithmetic implementation. However it is more complicated here because some of the data is stored in block RAMs and reads from these block RAMs need to be registered. The results from the *gamma-mul* stage are the input of this stage.

The eight results from the *gamma-mul* stage are added into four intermediate result matrices using four matrix-addition operations. These four matrix additions are processed within 48 cycles. However the *gamma-mul* stage begins writing to its result RAMs after 34 cycles, so the operands for the third and fourth matrix additions are copied into temporary storage to avoid a write before read data conflict between the two pipeline stages. Single ported distributed RAM is used for this temporary storage. Figure 6-b shows a diagram of how data is retrieved from the temporary storage RAMs, issued to the adders for addition, and the results stored back to the

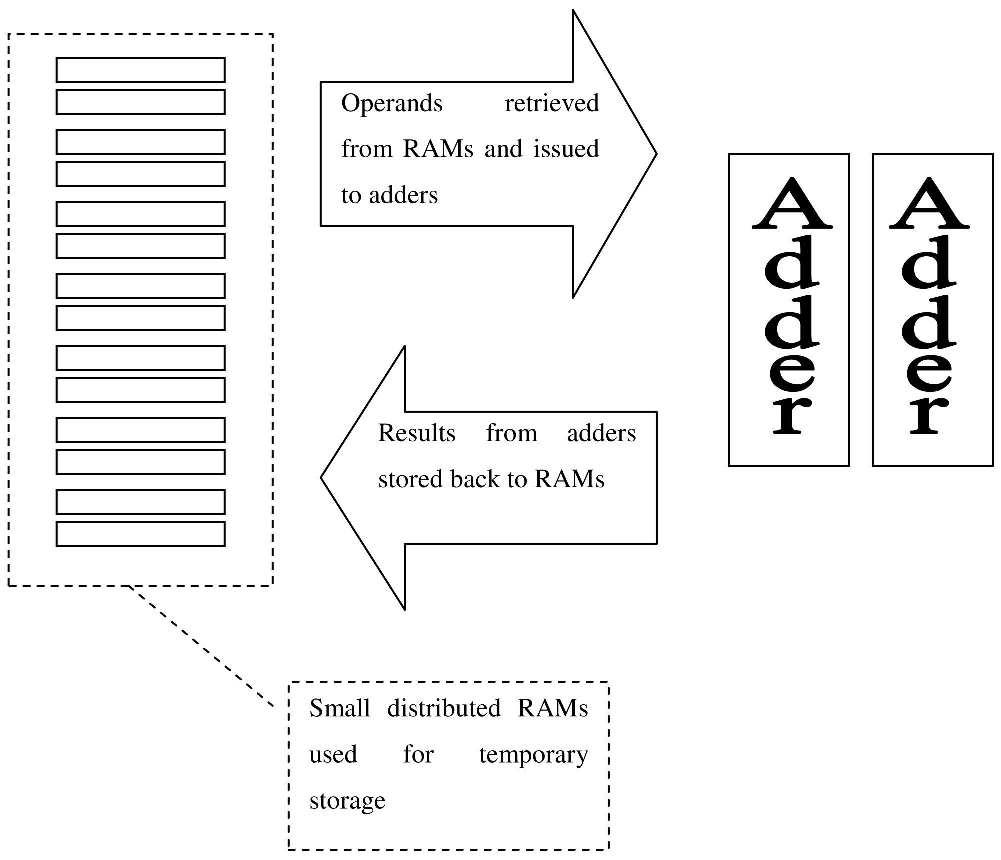

Operands retrieved from RAMs and issued to adders
Adder
Adder
Results from adders stored back to RAMs
Small distributed RAMs used for temporary storage

Figure 6-b. Diagram showing the operation of the matrix addition functions

temporary RAMs.

The matrix additions are performed using two separate operations. The first operation reads data from the operand storage and issues it to the pair of adder pipes used for this stage. There are two versions of this operation; one registers the output from the operand storage the other does not. The registered version is used when the operands are stored in block RAM the other for when the operands are stored in distributed RAM. Registering the output of the block RAMs takes an extra cycle to complete so the results are produced by the adder pipeline a cycle later. The retrieve operation includes an option to delay retrieval for a cycle which is used when operations are issued by the registered version of the issue operation.

Scale and subtract and the G5 operation

The results of the addition operations are scaled by a value called *kappa*, which is constant for a run of the application. The value of *kappa* determines how quickly the algorithm converges on a solution. The scale operation is performed by multiplying each number in the addition result matrix by *kappa*. The result is then subtracted from the data for the current site in the lattice, which has been pre-conditioned using the G5 operation. The G5 operation is very simple and does not require any arithmetic operations. To perform the G5 operation the operand matrix is copied directly into the result matrix, changing the sign of half the values. The sign of a number on a conventional processor is normally changed by subtracting it from zero, however it is possible to simplify this operation on an FPGA by simply flipping the sign bit of the floating point word. This is significantly faster than using a floating point operation.

The G5 operation is performed in parallel with the *gamma-mul* stage of the Dirac pipeline and the result is passed on to the addition stage. The result of the scale operation is then subtracted from the G5 result. The subtraction result is the final result for the current site in the lattice and is stored to off-chip memory in the next stage of the application pipeline.

Since the scale and subtract operations are always used together they are combined into a single operation. This operation issues the multiplications for the scale operation to a pair of multipliers. The multiplication results are then issued directly to the adder pipelines to perform the subtraction operation, eliminating any need for intermediate storage of the multiplication result data. The results of the subtraction are retrieved and stored in temporary storage using the same retrieve operation used by the addition operation.

To reduce the resource requirements of the design the scale operation is performed using the multipliers that are used for the *gamma-mul* blocks in the previous pipeline stage. Multipliers dedicated to the scale operation (as is done in the log arithmetic Dirac operator) would be a poor use of resources since they would be underutilised and would occupy a significant amount of FPGA resources. The multipliers used in the *gamma-mul* blocks are not used for the final 35 or so cycles of that pipeline stage, so they are available for use in the scale operation. To control use of these multipliers for the scale operation, the *gamma-mul* stage signals the addition stage once its use of the multipliers is complete. Thus the entire design uses ten adders and eight multipliers in total.

Putting the addition stage together

The addition issue and retrieve operations are combined with the scale-subtract operation inside a larger operation that runs the second stage of the application pipeline. This second stage operation has two parallel parts. The first runs the issues operations and the second runs the retrieve operations. The retrieve operations are delayed so that they retrieve data from the adder pipelines on the correct cycle. The second stage operation also ensures that no conflicts occur on the multiplier pipes shared between the scale-subtract operator and the *gamma-mul* blocks of the first stage.

6.3.4 Data storage and layout

Double Buffering of gl3 Operands

Off-chip data storage is a significant issue for the double precision Dirac operator. 384 64-bit floating point data words need to be read from or written to the off-chip memory for each site update. This must be done in a window of 180 cycles (the number of cycles taken by the *gamma-mul* stage). In order to avoid write before read conflicts on the *gl3* inputs to the *gamma-mul* stage, the *gl3* inputs must be double buffered.

Double buffering was not necessary in the logarithmic Dirac operator since all accesses by the *gamma-mul* stage to the *gl3* matrices are complete by cycle 72. This leaves a long window of nearly 100 cycles in which to read all the operands into the RAMs used by the *gamma-mul* stage. For the double precision Dirac operator this window is much smaller at only 27 cycles, which is insufficient to read in all the *gl3* operands. Only one multiplier, not two, is used in each *gamma-mul* block so it takes twice as many cycles to process all the operations involving the *gl3* operands.

In the double precision Dirac operator pipeline the *gl3* operands are buffered in on-chip block RAMs. These block RAMs are much larger than a *gl3* matrix, so there is a considerable amount of free space in the RAMs. The RAMs are dual ported and only one of these ports is required by the *gamma-mul* blocks. To double buffer the *gl3* inputs the block RAMs are divided into two logic spaces; “low space” and “high space”. The operand read pipeline stage uses one block RAM port and the *gamma-mul* stage uses the other port. Both pipeline stages alternate between the two spaces, and never use the same stage on the same pipeline iteration. This scheme double buffers the *gl3* inputs without using any extra FPGA resources.

Data Layout in Off-Chip Memory

Double buffering of the *gl3* operands allows all the *gl3* data to be retrieved within the available time whilst using a simple layout of the *gl3* matrices in off-chip memory. The *gl3* matrices are stored in two banks of SRAM with each 64-bit datum split in two. One half of each datum is stored in one bank and the other half is stored in the other bank. The real components of each number in the *gl3* matrices are stored in the evenly numbered memory addresses of the two banks and the imaginary components are stored in the odd numbered memory locations.

The bandwidth requirements for the *wfv* matrices are not as constrained as for the *gl3* operands. All accesses to the *wfv* operand matrices by the *gamma-mul* stage are complete by cycle 12 leaving 168 cycles in which to read all the operands in from off-chip memory. As a result double buffering was not required for the *wfv* operands. 216 64-bit words need to be read from one of the *Y* type matrices and 24 64-bit words written to another of the *Y* type matrices. Each of these matrices may be stored across 2, 4 or 6 SRAM banks. If stored across two then 216 cycles are required to read all the operands. If stored across 4 then 108 cycles are required and if stored across 6 banks then 72 cycles are required. The large *Y* type matrices are stored across four banks of off-chip memory, since only 168 cycles are available to retrieve the data.

6.3.5 Resource utilisation reduction

In the Xilinx Virtex-II FPGA that was used for this work the hardware multipliers and block RAMs are located in the same logic cell in the device. As a result they share the same ports to transfer data in and out of that cell. When a hardware multiplier in a cell is used in a design then the block RAM at that cell can only be used as a 16 bit wide RAM, because the ports are configured to be the width of the hardware multiplier's input, which is 18 bits. 72 hardware multipliers are used in the design by the floating point multipliers, so that meant that 72 of the available block RAMs could only be used as 16 bit wide block RAMs. A specially designed Handel-C struct was used to aggregate 4 of these 16-bit block RAMs into a single 64-bit block RAM. A set of macros were designed to handle reads and writes to these aggregated block RAMs.

6.4 Double precision conjugate gradient solver

The double precision conjugate gradient solver requires double precision versions of the *dot-product* and *matrix scale-add* operations. Data layout in memory determines how the operations are implemented. Double precision variables are 64 bits wide, so the ratio of memory bandwidth to calculation is doubled compared with single precision, making data layout very important. The *wfv* matrices are stored across 4 memory banks, using all six would increase bandwidth but would make the memory access hardware unfeasibly complex.

6.4.1 Double precision matrix add-scale operator

The *matrix add-scale* implementation streams data for one operand matrix from memory into a multiplier where it is scaled by a fixed value. The result is then added to the equivalent value from other operand matrix. Both matrices are stored in the same four off-chip memory banks so retrievals must be alternated between the two input matrices. The results of the operation are buffered into a set of block RAMs and the results are written out to memory when the block RAM buffers are full. Block RAMs at sites where the multipliers were already used were used for the buffer RAMs. These block RAMs were not used elsewhere in the design allowing a simple and efficient implementation of the matrix add-scale operator.

6.4.2 Double precision dot-product operator

The double precision dot-product operator uses a similar architecture to the one employed successfully for the logarithmic arithmetic version. Data is streamed directly from external RAMs into the multiplier to square it. The results of these squares are then accumulated using the architecture shown in Figure 5-f. The shorter latency of the double precision adder compared with the log arithmetic adder means there are fewer partial products to be accumulated which improves performance slightly compared with the logarithmic implementation. There are fewer partial products to be added together at the end, which reduces the depth of the reduction operation by one level.

6.5 Summary

The chapter has described the design and implementation of an FPGA based Dirac operator and conjugate gradient application using IEEE double precision floating point arithmetic. The double precision FPGA designs were subject to many more design constraints than the designs that used logarithmic arithmetic. The logarithmic adders are large and have long pipeline latencies and the logarithmic multipliers are very small and have short latencies. Consequently the design priority for logarithmic designs is to make good use of the available adders. In comparison the IEEE double precision adder and multiplier are similar in size and have similar pipeline latencies. Thus it is important to make good use of both types of unit. Also since double precision data words are twice the size of single precision data words the data bandwidth requirements of the double precision designs are doubled compared with the single precision logarithmic designs.

As a result of these design constraints the double precision designs are much more constrained than the logarithmic designs. Nonetheless the designs are efficient and return excellent performance for both the Dirac operator and for the full conjugate gradient solver application. These designs show that FPGAs can be used successfully for lattice QCD applications using double precision arithmetic.

The computational requirements of lattice QCD are very significant and can not be met by a single processor or FPGA. Consequently it was decided to investigate the possibility of multiple FPGA systems for lattice QCD. I did this by implementing a dual-FPGA version of the logarithmic arithmetic Dirac operator described in Chapter 5. This dual-FPGA Dirac operator is the subject of the next chapter.

Chapter 7

Dual FPGA Dirac Operator

The previous two chapters have described FPGA designs that implement key lattice QCD algorithms on FPGAs using both logarithmic arithmetic and IEEE double precision floating point arithmetic. These implementations show that FPGAs can return good performance for lattice QCD simulations. However neither set of designs use more than a single FPGA. Generating a single scientific result using lattice QCD simulations requires in the order of 6.6×10^{15} floating point operations. It would take over two months to do this using double precision FPGA designs described in the previous chapter, and given that many such scientific results need to be generated for a single problem, it can be seen that a single FPGA will never be sufficient to meet the computational requirements of lattice QCD.

In order to demonstrate whether an FPGA based solution has potential to meet the demands of lattice QCD a multiple FPGA version of the Dirac operator was implemented. This dual FPGA Dirac operator demonstrates that two FPGAs can be successfully applied to one use of the Dirac operator. This result shows that multiple FPGA systems have potential for lattice QCD. The dual FPGA Dirac operator is based on the log arithmetic Dirac operator detailed in Chapter 5.

7.1 Partitioning the Dirac operator algorithm

The log arithmetic Dirac operator is implemented across two FPGAs. The Dirac operator is highly parallelizable. Each site update within a single sweep is independent of the other updates, but the sweeps are dependent, so one sweep cannot begin until the previous one has finished. Consequently the site updates for a sweep can be performed easily on multiple processors. After the sweep has finished, or during the sweep, the results for each site update are sent to the other processors for use in the next sweep of the lattice. Half the site updates are performed by each FPGA, with the lattice split in two along the time axis. One FPGA updates the lower half of the sites and the other FPGA updates the upper half.

The operator pipeline for the Dirac operator is unchanged for the multiple-FPGA version, except that the result-write now obtains the SRAM semaphore before writing out any results to the SRAM banks. Thus, when the Dirac operator pipeline is not writing data to the SRAM banks, the communications package can write data received from the other FPGA. Consequently calculation never needs to wait for data to be sent from the other FPGA and communication and calculation are fully parallelised.

For the dual Dirac operator it is assumed that all the results produced by one FPGA must be sent to the other FPGA. This is the case for small problem sizes. For large problem sizes only a subset of the results calculated *must* be sent to the other FPGA. However by assuming that all results must be sent for all problem sizes the scalability of the dual-FPGA system is more rigorously tested.

The Dirac operator pipeline produces one result every 168 cycles. Each result consists of 24 words, which is 96 bytes when single precision arithmetic is used. Therefore minimum bandwidth of 0.57 bytes per cycle is required to transmit the data, ignoring signalling overhead. However the flow of data between the FPGAs must be controlled, which requires extra bandwidth and it may not be possible to transmit data on every clock cycle. Therefore some extra bandwidth is required so available bandwidth of 2 bytes per cycle should be sufficient.

7.2 Communications requirements of the Dirac operator

Using multiple processors for a single lattice QCD calculation is a well established technique. The core Dirac operator algorithm is very well suited to such architectures: it is highly parallelizable, it has reasonable communications bandwidth requirements and, importantly, requires only nearest neighbour communication between processing nodes. Nearest neighbour communication is simpler to implement than any-to-any communication and is more efficient as it eliminates the complexity of any-to-any communication systems.

A multiple processor lattice QCD machine needs a number of things to have good performance for a large number of processors. They are:

- Very low latency communications
- Good bandwidth
- Ability to parallelise computation and communication
- Ability to pre-fetch data from memory

Bandwidth is important for multiple processor lattice QCD systems but low latency is more important. Communications latency determines the scalability of a system when applied to lattice QCD problems. Scalable systems can apply many processors to a single problem. If communications latency is high then performance can be lost since the processing nodes spend a lot of time waiting for data to arrive from other processors. Scalable systems, such as the QCDOC system (see section 3.2.3) avoid such performance loss through a combination of low latency communications systems and parallelised communications.

Scalable systems allow the lattice to be divided into very small portions so that each node performs the calculation for a small number of lattice sites. Systems such as QCDOC [P. A. Boyle '05] and apeNEXT [Belletti '06] are custom ASIC based systems with communications architectures tailored to the requirements of lattice QCD. The systems are very scalable; each node in the QCDOC can operate effectively on as few as 16 sites in a lattice.

PC based clusters, as described in [Holmgren '05b], are generally not as scalable since they suffer from higher communications latencies. Each PC cluster node needs to operate on at least one thousand sites or overall performance for the system is poor. Performance is improved significantly by having each node operate on at least several thousand sites. The priority in designing the communications system for the dual FPGA implementation was to minimise latency in the system whilst preserving bandwidth.

7.3 Hardware Infrastructure

The Alpha-Data ADM-XRC II prototyping boards that were used for the log arithmetic and double precision applications are described in section 2.1.1. The ADM-XRC II boards consist of a PCI carrier card which can accept up two PCI Mezzanine Cards (PMCs). The PCI carrier card connects the PMC cards to the host computers PCI bus. The carrier card also connects the FPGAs on the two PMCs with 64 pin to pin wires. For the dual FPGA implementation a carrier card with two ADM-XRC II PMC cards mounted on it was used. A low latency architecture was designed which uses the interconnection provided by the carrier card to communicate data between the two FPGAs.

7.4 Existing Inter-FPGA Communications Systems

A variety of different systems exist for communicating data between two FPGAs. An example of such systems is the Xilinx SelectLink system, which is discussed here. The full name of this system is the Virtex-II SelectLink Communications Channel. It is a freely available system for communicating data between two Xilinx FPGAs [Logue '05]. The system provides a low latency, high bandwidth, unidirectional, communication system that transmits data from one FPGA to another. Xilinx provide an internet tool that generates VHDL or Verilog source for the system that is customised to a designer's requirements.

The SelectLink system is a source synchronous communications package that appears, to the user logic on the transmitting and receiving FPGAs, to be a single FIFO between the two FPGAs. Two of these channels can be used to provide bidirectional communications between the two FPGAs. This is shown in Figure 7-a. The user logic writes data to be transmitted to a FIFO in the transmitter block. The SelectLink transmitter logic then reads the data from this FIFO and transmits it to the other FPGA. The data is transmitted along with the clock of the transmitter. The

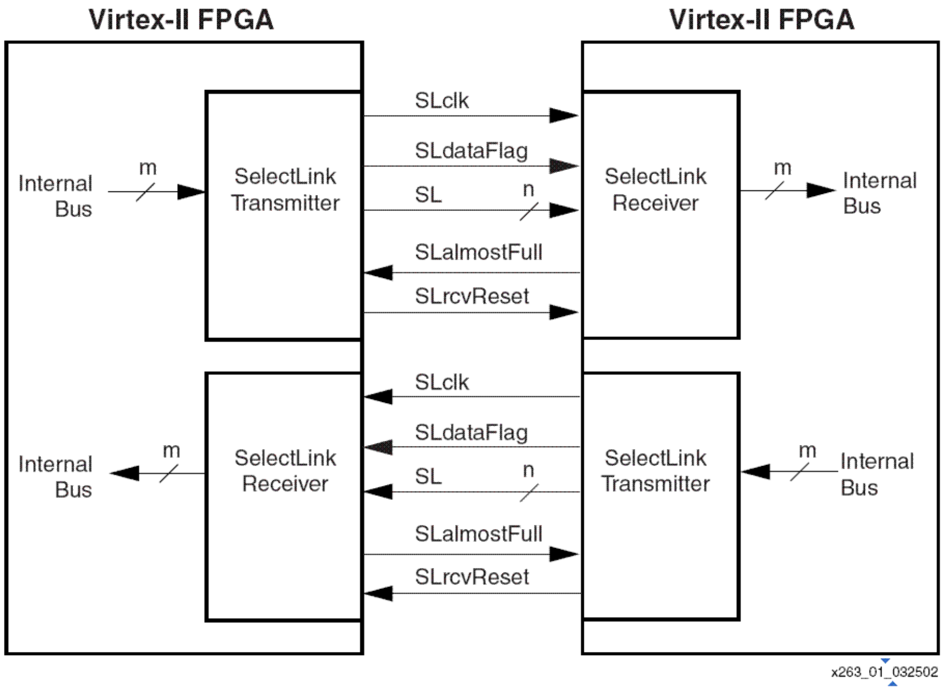


Figure 7-a. Xilinx Virtex-II Communications Channel [Logue '05]

receiver logic on the receiving FPGA is clocked using this clock, ensuring that the receiver logic stays in phase with the transmitter. The receiver then writes the received data to a FIFO. The user logic on the receiving FPGA can then read the data from this FIFO.

The SelectLink system allows the designer to choose one of a number of data transmission standards to handle transmission of data across the wires. Which signal standard is chosen depends on the nature of the electrical connection between the FPGAs and on the data bandwidth requirements of the design. If the interconnect consists of short on-circuit board wires with good signal integrity, then a simple, single ended protocol is sufficient for moderate data rates. If longer wires are used, perhaps between circuit boards, or if higher data rates are required, then a more sophisticated signalling standard is needed. Using sophisticated signalling standards with short on-circuit board wiring allows the system to achieve very high clock rates and hence achieve very high bandwidth.

The analysis of the bandwidth requirements of the logarithmic arithmetic Dirac

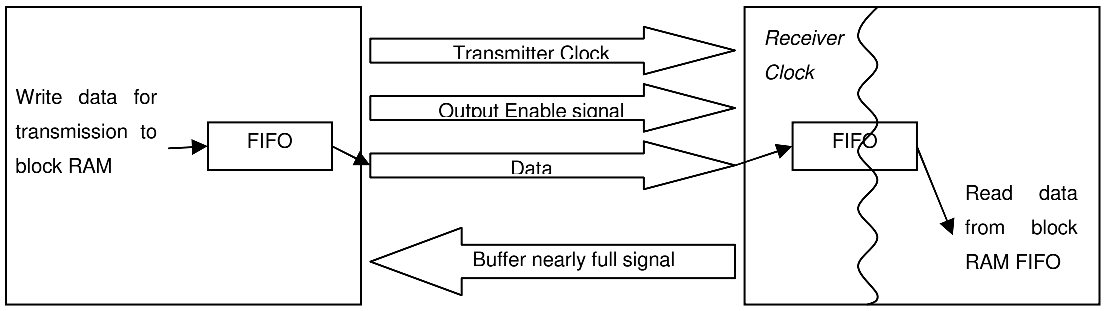

Write data for transmission to block RAM
FIFO
Transmitter Clock
Output Enable signal
Data
Buffer nearly full signal
Receiver Clock
FIFO
Read data from block RAM FIFO

Figure 7-b. Diagram of source synchronous communications system

operator, see section 7.1, showed that the SelectLink system would be sufficient for the bandwidth requirements of a dual-FPGA version of the operator. Unfortunately the SelectLink system proved to be incompatible with the off-chip memory access logic in the Dirac operator design. The SelectLink system is intended to be used as part of a VHDL or Verilog design and it proved to be incompatible with parts of the Handel-C design.

No other communications system was available that met the needs of a dual FPGA based Dirac operator. Consequently a Handel-C based system was developed. This design is a simplified version of the SelectLink architecture; it does not implement some unnecessary and complex aspects of the SelectLink architecture. In particular, the SelectLink architecture transmits data using Double Data Rate (DDR) signalling, where data is transmitted on both the rising and falling edges of the clock, to improve bandwidth. DDR is efficient but very complex to implement and so it was omitted since the extra bandwidth was not needed.

7.5 Source Synchronous Communications

The overall architecture of the uni-directional communications system is shown in Figure 7-b. I decided against a bi-directional system since both FPGAs need to send a similar amount of data, so it makes more sense to have two simple uni-directional systems instead of one, complex, bi-directional system. In a bi-directional system only one FPGA may communicate at a time, and controlling which FPGA gets access to the communications is a complex task. Having two uni-directional systems gives bi-directional communication with less complexity and sufficient performance for the Dirac operator. Another useful benefit of using two uni-directional communications systems is that data may be sent in both directions at the same time.

Communications systems can be either synchronous or asynchronous; synchronous systems use a clock to determine when to sample the data bus whilst asynchronous systems use *output enable* and *acknowledge* signals to handshake data across the data bus. Asynchronous systems are quite simple but they have poor bandwidth since they require several cycles to transmit a single datum. Synchronous systems can usually transmit one datum per cycle, but they must ensure that the receiver's clock stays in phase with transmitter's clock. There are a number of ways to do this including: sending the transmitter's clock with the data, clocking both FPGAs using the same clock or encoding the clock into the data in some way.

The low bandwidth of asynchronous communications systems makes them unsuitable for the needs of a dual FPGA Dirac operator. It is not possible to clock both PMC cards on the PCI carrier card using the same clock, so a synchronous protocol where both FPGAs are clocked by the same clock was not possible. It was decided to implement a source synchronous system where the transmitter's clock is transmitted along with the data.

The uni-directional communications system, like the Xilinx SelectLink system, is a source synchronous system where the transmitter's clock is transmitted with the data. The receiver is clocked using this transmitted clock, ensuring that the data is read correctly from the transmission lines. The clock and data arrive at the receiver of the other FPGA in phase allowing more reliable operation of the communications system. This is a well established technique that is used for the DDR SDRAM (Double Data Rate Synchronous Dynamic RAM) found in most current PC machines [JEDEC '05]. A further advantage is that this system can easily be extended to systems with more than two FPGAs.

In the system the transmitter transmits data to the other FPGA over a 16-bit wide data bus. It also transmits an Output Enable signal to tell the receiver that there is valid data on the bus. The receiver samples the data bus only when Output Enable is asserted. The transmitter also transmits its clock to the receiver. The receiver is clocked using this clock ensuring the data is sampled correctly. One 16-bit data word can be transmitted on every cycle thus ensuring excellent bandwidth.

The actual operation of the system is similar to the operation of the SelectLink system. The application logic on the transmitting FPGA writes data to a transmit FIFO queue. The transmitter then reads the data from the FIFO and transmits it to the receiving logic on the receiving FPGA. The receiving logic then writes the data to a receive FIFO. The user logic can then read the data from this FIFO at any stage. This system appears, to the FPGA designer, to be a FIFO between the two FPGAs.

If a lot of data is transmitted to the receiver but not read by the application logic on the receiving FPGA then the FIFO queues can get backed up. To prevent data loss the receiving FPGA transmits an almost full flag back to the transmitter. The receiver flag asserts this flag when the receiver FIFO is three quarters full. When this flag is asserted the transmitter stops reading data from the FIFO connected to the application logic but sends any data that has already been read from this FIFO. The almost full flag is cleared when the receiver FIFO is less than three quarters full; when this happens the transmitter can start transmitting data again. When the transmit FIFO is nearly full the application logic is prevented from writing to it so the application design cannot rely on the FIFO being available. These two mechanisms control the flow of data from the transmitting FPGA to the receiving FPGA.

7.6 Implementing the design

7.6.1 FIFO queues and Handel-C

Handel-C FIFO

The FIFO queues themselves were a major obstacle to implementing the communications system. Handel-C provides a FIFO construct which is based on the blocking communications channels that are used for cross clock domain communication and for synchronising two parallel sections of a Handel-C design. These channels can be made non-blocking by adding a *fifolength* = *x* specification to them, where x is greater than zero. Such channels are non-blocking for both reads and writes, except when reading an empty FIFO, or writing to a full FIFO.

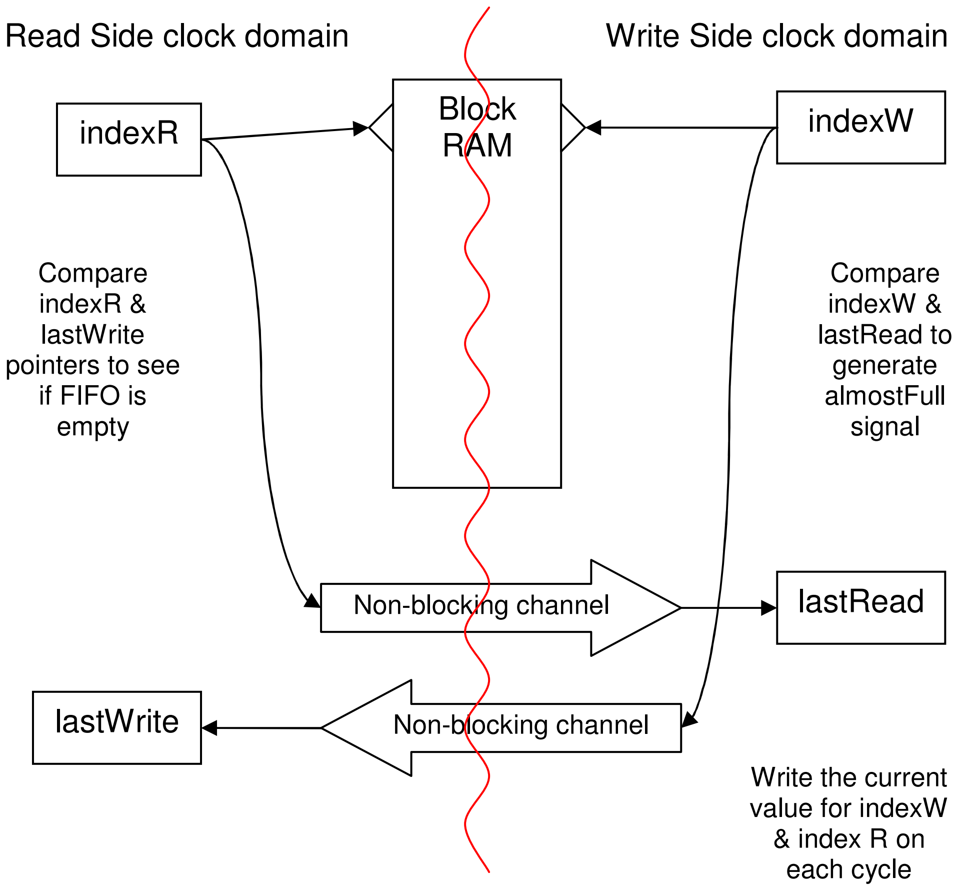

Read Side clock domain
Write Side clock domain
indexR
Block RAM
indexW
Compare indexR & lastWrite pointers to see if FIFO is empty
Compare indexW & lastRead to generate almostFull signal
Non-blocking channel
lastRead
lastWrite
Non-blocking channel
Write the current value for indexW & index R on each cycle

Figure 7-c. Diagram of FIFO used in the source synchronous communications system

Unfortunately, these FIFO channels do not provide any indication of when the FIFO is *nearly* full. They merely block when they are full. This is a major drawback, since the communications system is pipelined and if the receiver FIFO blocks then any data in the pipeline is lost.

To solve this problem, whilst still using the Handel-C FIFO, I created logic to generate almost full flags for the Handel-C FIFO by tracking the number of reads and writes to the FIFO. In order to fully pipeline communications system I needed to ensure that FIFO reads never blocked. Handel-C also provides an ability to test a channels readiness using the *prialt* statement. I used *prialt* to test the channels before reading from them to ensure that the communications system never blocked due to an empty FIFO. Unfortunately I found that the *prialt* statement does not work for channels that cross clock domains, as the FIFO channels in the communications system do. Consequently I had to abandon using the Handel-C channels and write the own FIFO instead. In earlier versions of the Handel-C tools *prialt* did not work for channels that cross clock domains, however Celoxica claims that this problem has been fixed. Unfortunately this was found not to be the case.

FIFO in the Source Synchronous Communications System

A FIFO is a well established hardware construct and there are several solutions available including one from the OpenCores initiative and one in the Xilinx Core-Generator library. The development environment for Handel-C, the DK suite, can use third party IP cores for hardware designs. However, simulating designs that use these cores is very difficult; DK cannot use the simulation files that are supplied with the cores since they are targeted at VHDL or Verilog simulators. Instead users must write their own simulator libraries which, assuming one can find how to do this, is as much trouble as implementing the core from scratch in Handel-C. It is extremely difficult to create Handel-C designs without using the simulator so I decided to write the own FIFO from scratch in Handel-C, which allowed comprehensive debugging of the designs using the simulator.

Figure 7-c shows the design of the FIFO implementation. The FIFO needs to be able to cross clock domain boundaries with the read side in one domain and the write side in the other domain. Block RAMs can easily transfer data between clock domains simply by writing data to the block RAM in one domain using one port and reading in the other domain using the other port. Each side of the FIFO needs to know when it is able to read or write data, so each side needs to know the index in the block RAM of both the last read and write. The read side sends the current value of indexR (the read index) to the write side on every cycle using a non-blocking channel. Likewise the write side sends the latest value for the write pointer to the read side. The channels transmit the values across the clock domain boundary with a minimal chance of data corruption.

Before performing a read, the read side checks *indexR* and *lastWrite* to ensure that the block RAM location to be read holds valid data. If *indexR* is less than *lastWrite* then the data to be read is valid. Wrap around within the FIFO, where new data overwrites older unread data is prevented since the FIFO will not accept new data if it is more than 75% full. Before performing a write, the write side checks the *almostFull* flag. If this flag is asserted then a write may not be attempted. The *almostFull* flag is updated on every cycle, and is set if the gap between *indexW* and *lastRead* is greater than three quarters of the size of the FIFO. Leaving a quarter of the FIFO empty ensures that there is always space to store any data in the pipeline supplying the FIFO.

7.7 Communications protocol

In order to use the communications system for a dual FPGA Dirac operator a communications protocol implemented to manage data transfer between the two FPGAs. The protocol also provides the ability to synchronise the two FPGAs. Synchronisation is necessary to ensure that the two FPGAs are both working on the same part of the algorithm at the same time.

The system is designed to be extendable, allowing it to be used to implement other operators, such as the dot-product operator, over two FPGAs. All messages begin with a control word. The control word is shown in Figure 7-d. Each command type has a unique message-type number. The control word also provides extra space to send a small amount of data along with the message type number.

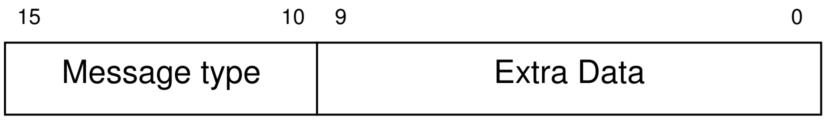


Figure 7-d. Communications protocol control word

The receiving FPGA’s application logic uses a state machine that reads the received data from the receiver FIFO. When a control word is read from the FIFO, the state machine inspects the control word and starts the appropriate handler for the message type. This handler then processes the remainder of the message. Different message types have different lengths, however all messages of a given type are the same length. Two message types are implemented, but more types could be added.

- Send *wfv* matrix
- Send synchronise message

Sending and Receiving *wfv Ma*trices

The *send wfv* message allows a single *wfv* matrix to be sent from one FPGA to the other. Information about which pair of SRAM banks the wfv matrix should be stored in is sent in the spare space in the command word. The first eight bits of the address offset into these banks is also included with the code word. The next datum in the message contains the site number of the *wfv* matrix. The receiving FPGA uses this information to decide where to store the incoming *wfv* matrix. The *wfv* matrix is packed into the message body after the two control data. Each communications data word is 16-bits wide so the 32-bit wide elements of the matrix are split in two with the higher 16-bits sent first followed by the lower 16-bits.

On receiving a *send wfv* message the receiving FPGA decodes the memory bank, address offset and site number information. The *wfv* matrix is then read from the receiver FIFO and written to the appropriate location in the off-chip SRAM.

Synchronising the FPGAs

A synchronisation mechanism is required to ensure that the two FPGAs only work on the same iteration of an operation at any given time. The two FPGAs synchronise at the beginning and end use of the Dirac operator. The dual FPGA Dirac operator is run many times in succession in order to collect valid performance data for it, and the two FPGAs must always work on the same iteration of the Dirac operator for the performance results to be valid. When the operator is used in a full application it is used in conjunction with other operations. Each call of the Dirac operator must complete before any other operator may be called.

In order to synchronise, a barrier type synchronisation point is implemented by both FPGAs sending a synchronise message (either *StartCalc* or *EndCalc* depending on whether the critical section is starting or finishing). Immediately after sending a synchronisation message the user logic reads from a blocking channel connected to the receiver state machine. This forces the user logic to stall until an equivalent synchronisation message is received from the other FPGA. When a synchronisation message is received, the receiver's state machine writes to this blocking channel, allowing the application logic to continue. The blocking channels ensure that one FPGA cannot pass a synchronisation point until the other FPGA has reached that same point.

Controlling access to the SRAM banks

Access to the pairs of SRAM banks used to store the large *Y* type matrices is controlled using semaphores. No part of the application accesses the SRAM banks without first possessing the semaphore for the appropriate bank pair. Handel-C includes a semaphore construct which is used here. Only one part of the design may hold a particular semaphore at any given time. Any requests for a held semaphore will stall until the semaphore is released. This prevents the communications package conflicting with the result write of the Dirac operator and permits each FPGA to send results to the other FPGA during the calculation. The other FPGA can then write the results to the SRAM during the calculation, when the SRAMs are not being used by the application logic, allowing full parallelisation of calculation and communication.

7.8 *Communication difficulties*

7.8.1 Errors in the communication system

It was not possible to fully eliminate all transmission errors from the communications package; a very, very small fraction of data is transmitted incorrectly. It was very difficult to trace the cause of the error due to the limitations of using Handel-C for this type of design work. The Handel-C simulator cannot simulate inter-FPGA communication accurately, so the only way to diagnose the problem was to repeatedly place and route designs, using the results to try and determine where any problems lie. This is a clumsy approach, and since it was not possible to inspect the internal signals in the FPGA, it was not a very good way of finding the problem with the dual-FPGA design.

After much investigation I decided that the transmission errors are caused by a timing problem; if a design is placed and routed twice, and the two resulting bit files run, then errors displayed by the two runs are different. This behaviour is consistent with a design suffering from timing errors. Further evidence that the problem was timing related was that the problem only appeared on large complex designs; simple designs worked perfectly. This is consistent with a timing related error.

With the current system, if the messages involved in a synchronisation are transmitted incorrectly then the system will stall while one of the FPGAs waits for a message that will never be received. The unreliability of the communications system therefore limits how long the dual-FPGA system can run for; if the system is too unreliable then the system will stall before valid performance data can be collected. By repeatedly placing-and-routing the design I was able to create a version of the communications system that runs the dual-FPGA Dirac operator for several seconds without error. By sending an inverted clock with the data I found that the system was sufficiently stable to allow performance results for the dual-FPGA Dirac operator to be collected.

Since the purpose of the dual-FPGA operator is to prove the scalability of FPGA based systems, I decided that this result would be sufficient. The aim was not to produce a fully reliable production quality system, but was to show the scalability of a dual-FPGA based system for lattice QCD calculations. All potential solutions to the problem would involve a very significant amount of work, given that it had taken several months to create a design that could be used to get performance results, I decided that this design was sufficient.

7.8.2 Solving the communication problems

A possible solution to this problem is to use a Digital Clock Manager (DCM) to ensure that the receiver's rising clock edge falls in the middle of the stable window of the incoming data. The data window is the portion of the clock cycle where the incoming data is stable and can be correctly sampled. To ensure that the clock is phased correctly the DCM monitors the data transmission lines, then, when the system is started, the transmitter transmits a "training pattern" on the data lines which the DCM on the receiver uses to determine where the middle of the data window is. The DCM then phase shifts the receiver's clock to fall in the middle of this data window.

VHDL and Verilog development environments can perform timing accurate simulations of designs that use features, such as DCMs, that are embedded in modern FPGAs. VHDL or Verilog simulators allow designers to inspect any signal in a design, and then accurately compare the signal's transitions to other signals in the design. Such a simulation can be repeated after each of the place and route stages (translate, map and place-and-route) to ensure that the design remains correct. Debugging DCMs, and similar components, without such a sophisticated simulator is *extremely* difficult. It is impossible to simulate DCMs in Handel-C; a design using a DCM can only be tested in hardware, where it is not possible to inspect the signals attached to the DCM. The DCM either works or it does not. The Handel-C simulator is targeted at on-chip algorithmic designs; it is not suited to designing complex off-chip communications systems that require complex clocking logic. The only viable debugging option available was to test the designs in hardware, using the relative correctness of the results to gauge whether the change was a success.

For this solution to work the communications system would need to be re-implemented using a different hardware design language, such as VHDL. Also the Dirac operator would require substantial redesign to eliminate conflicts between the off-chip SRAM logic and the communications package which prevented the SelectLink system from working.

An alternative solution would be to change the communication system so that it is tolerant of unreliable data transmission. This could be done by adding parity bits to the data wires. The parity of each data word to be sent would be calculated by the transmitter before transmission. The receiver could then use this parity information to detect when data has been transmitted incorrectly, and request re-transmission of that data. However this system would require a significant amount of work. The transmission protocol used for the dual-FPGA operator is very simple, and has no ability to request retransmission of data. Adding this functionality would be complex and would require a significant redesign of the communications system.

7.9 Summary

The dual-FPGA Dirac operator described in this chapter demonstrates that FPGAs have the potential to be a scalable platform for lattice QCD simulations. The communications system used here is a widely used mechanism which delivers sufficient bandwidth and, more importantly, low latency inter-FPGA communications. This system is used to implement a dual-FPGA version of the log arithmetic Dirac operator that is described in Chapter 5. The dual-FPGA version delivers performance that is nearly twice the performance of the single FPGA version, by parallelising nearly all communication with calculation.

The next chapter presents performance results for the three categories of design presented in this work, including the single FPGA logarithmic arithmetic designs, the IEEE double precision designs, and the dual FPGA Dirac operator described in this chapter. The performance of these designs is discussed and compared with the performance of a range of existing lattice QCD machinery.

Chapter 8

Results

This chapter presents results for the logarithmic arithmetic designs described in Chapter 5, the IEEE double precision arithmetic designs described in Chapter 6, and the dual-FPGA Dirac operator described in Chapter 7. The results of these designs are compared with a variety of systems that are used for lattice QCD simulations. These lattice QCD systems are described in detail in section 3.2. Many of these systems, including the two custom ASIC based supercomputers, are the result of significant research effort. The performance of the FPGA based lattice QCD operators is compared with these machines. This comparison delivers significant insights into the usefulness of FPGAs for high performance computing machinery.

8.1 Correctness testing

All the single FPGA designs presented in this thesis have passed extensive testing for correctness. The procedure is as follows.

- An adapted version of the original application is used to randomly generate input data and a corresponding correct result for the FPGA designs.
- The input data is used to run the FPGA design, and the results are saved.
- Then a small program is used to compare the FPGA result to the correct result. This program compares each element of the correct result with the equivalent FPGA result. If the difference between any two results is greater

than a certain threshold then the FPGA design is incorrect.

The lattice QCD designs do not require the input data to conform to any specific rules; randomly generated input sets are sufficient to test the designs. This was determined after discussions with application experts, who were satisfied that randomly generated input data is sufficient to rigorously test the application.

The results from the correct application rarely exactly match the results from the FPGA. The correct results are calculated on an Intel Pentium 4 processor using double precision floating point, whilst the FPGA designs use 32-bit single precision logarithmic arithmetic or 64-bit double precision floating point arithmetic. Internally the Pentium 4 converts all floating point numbers to an 80-bit extended double precision format to increase the accuracy of its floating point calculations. The results are then converted back to standard double precision floating point for storage. As a result of this extra precision the Pentium 4 results usually differ from those generated by either class of FPGA design.

For the Dirac operator designs the difference between correct FPGA results and the correct results is not large; approximately 1×10^{-7} for the logarithmic designs, and 1×10^{-13} for the double precision designs. In both cases the difference is very small. The differences for a full run of the conjugate gradient application are larger since the data is reused many times. However each component of the conjugate gradient application was extensively tested individually and found to be correct. The full applications were also tested by only iterating the main loop once, which reduced the accumulative effect of the lower precision used in the FPGA designs. Finally the conjugate gradient applications were tested for full runs involving many iterations of the main loop, and the results were found to be close to the correct results.

8.2 Performance measurement methodology

The performance results quoted in this chapter do not include the time required to load the application data on to the FPGA prototype board, nor do they include the time required to convert the input data to the logarithmic domain for the logarithmic arithmetic designs. The designs described in this work are intended to investigate the viability of FPGAs for high performance computing applications. They are not proposed as lattice QCD accelerators for a PC type machine.

All results that I present here are for fully placed and routed designs, running in hardware, on the Alpha-Data ADM-XRC II board. All the designs have been placed and routed at high effort levels, and have been extensively optimised to maximise clock rate. The clock rate figures shown for each design are the maximum possible clock rates for these designs, as reported by the Xilinx place and route tools.

The Xilinx place and route tools output an FPGA configuration file, called a bit file, which when downloaded to the FPGA determines the FPGA's behaviour. I have written software code, using the application programming interface supplied with the card by Alpha-Data, which runs on the computer hosting the ADM-XRC II card. This code downloads the bit file to the FPGA and, using the PCI bus, transfers all the operand data for the application to the card's SRAM banks. Once the data transfer is complete the FPGA design is ready to start. The host computer signals the FPGA, using the status register, to start it running. The status register is a blocking communications system, between the host and the FPGA, that can be used to synchronise the host and FPGA. Immediately before signalling the FPGA the host computer records the current value of the system clock. Once the FPGA has finished running, it signals the host program, and immediately after receiving the signal the host program records the value of the system clock for a second time; the difference in the recorded times is the execution time of the program, which is used to calculate the performance figure.

The run-time of a single call of the Dirac operator is too short to measure accurately since it runs for in the order of 20ms. In order to get an accurate measurement the operator is run for at least several thousand iterations, so the overall run time is in the order of two to three minutes. This also reduces the time spent signalling the start and end of the calculation to a small fraction of the overall run time. The number of iterations used depends on the problem size; to maintain a reasonable overall run time, larger problem sizes run for fewer iterations.

The conjugate gradient application runs for long enough on its own so this technique

Operator	**Floating point operations per site**
Dirac Operator	2616
Matrix Add-Scale	48
Dot-Product	48
Conjugate Gradient	5472

Table 8-a. Floating point operations per site for the lattice QCD operators.

is not necessary when measuring its performance. The conjugate gradient application applies each operator to the lattice a number of times: twice for the Dirac operator, three times for the matrix add-scale operator and twice for the dot-product operator. The operators are applied repeatedly in a particular sequence, see Figure 3-d, until the difference between the results of the last and second last iterations falls beneath a pre-set threshold; the difference is evaluated by comparing the results of the applications of the dot-product operator. Consequently, the number of iterations of the conjugate gradient operator is variable, so the FPGA design counts the number of iterations and returns this value to the host computer when the solution has been reached.

$$P = \frac{F \times N \times I}{T} \tag{18}$$

Performance figures for lattice QCD machines are normally quoted in Floating Point Operations per Second (FLOPS). Most machines used for lattice QCD are capable of several hundred MegaFLOPS (MFLOPS), or even over one GigaFLOPS (GFLOPS) per processor. All the implemented lattice QCD operations - the Dirac, dot-product and matrix add-scale operators - perform a fixed number of floating point operations per site, shown in Table 8-a. In order to determine the floating-point performance of the operators, the number of floating point operations that each operator performs for each site was counted. This data was used with (18) to calculate the floating point performance of each operator, where P is the performance, F is the number of

Operation	Clock Rate (MHz)	MFLOPS	FP Ops per Cycle
Dirac Operator	85	1320	15.5
Conjugate Gradient	85	1050	12.35

Table 8-b. Performance of the log arithmetic designs

arithmetic operations performed per site, N is the number of sites and I the number of iterations performed.

8.3 Log arithmetic implementation results

Table 8-b shows the performance for the Dirac operator and the conjugate gradient solver implemented using log arithmetic. The results shown in Table 8-b are for designs placed and routed for a Xilinx Virtex-II XCV6000 speed-grade 6 device. The performance for the core Dirac operator is very strong at 1320 MFLOPS and the performance for the conjugate gradient application is similarly strong at over one thousand MLFOPS.

Clusters of PC based computers are popular for lattice QCD simulations; this field of research is described in section 3.2.2. Generally lattice QCD simulations on PC clusters are run using single precision arithmetic since a PC's single precision performance is usually twice its double precision performance for lattice QCD. Log arithmetic is equivalent to single precision IEEE floating point so I compare the log arithmetic designs to the performance of PC processors. Other systems currently used for lattice QCD, such as QCDOC and apeNEXT, run double precision arithmetic. The log arithmetic implementations are not compared to these systems as it would not make a valid comparison.

The performance of PC machines for lattice QCD is very much a moving target, new processor designs, interconnect systems and memory buses are constantly improving performance for these systems. Consequently I present results for two systems. The

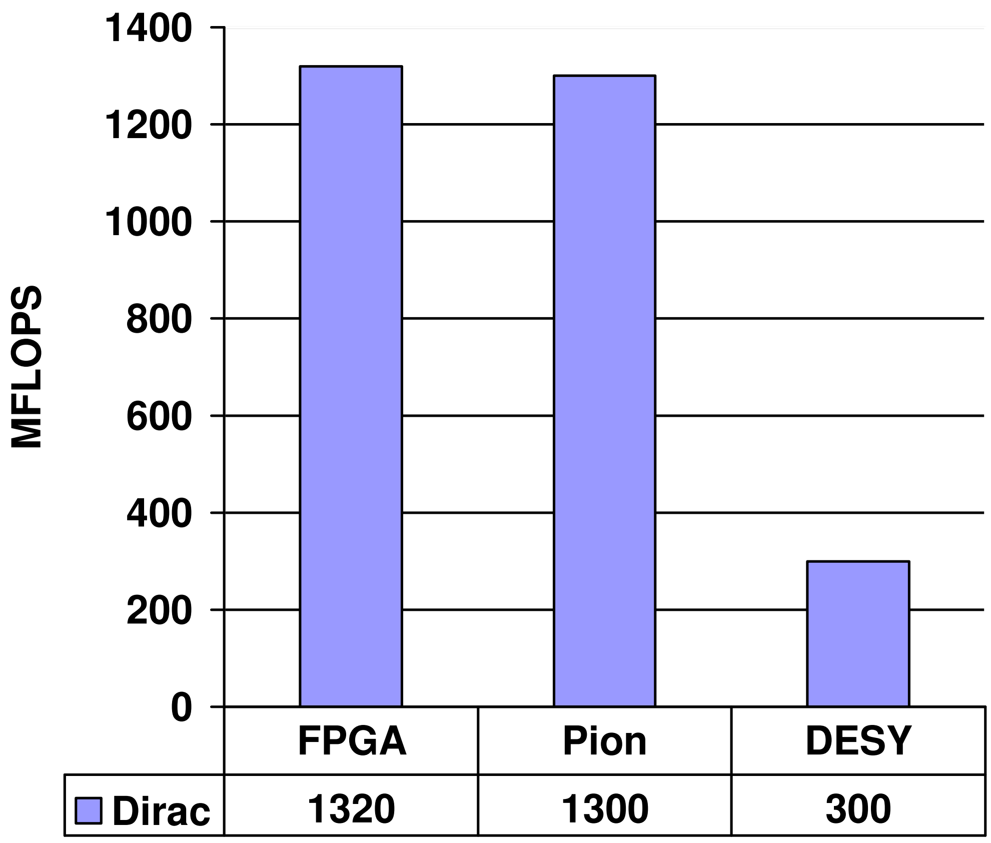


Figure 8-a. Performance of log arithmetic FPGA design and comparable systems

per-processor performance of PC's is much higher for single processor systems than for multiple processor clusters. Since the high single node performance cannot be obtained for real calculations I compare the performance of the designs with the performance of PC processors used within clusters of 16 CPUs, for a moderate per-CPU problem size of around 1000 sites per node. The delays caused by the communications systems can degrade performance by as much as half, although more recent communications technologies have reduced this effect [Wettig '05].

There are many different PC clusters used for lattice QCD; I compare my results with the Pion cluster described in [Holmgren '05a], which was commissioned in June, 2005. Pion consists of 520 nodes and each node consists of a 3.2 GHz Intel Pentium 4 processor with an 800MHz bus to main system memory. The nodes are connected using the Infiniband interconnect system, which is a recently available high bandwidth, low latency interconnect. Pion is, at the time of writing, the most sophisticated PC cluster used solely for lattice QCD calculations. Results are also shown for an older system, the DESY system, constructed using Intel Xeon 1.7 GHz processors and Myrinet interconnect, described in by Gellrich et al [Gellrich '03]. This system is about 4 years older than the Pion system, and is used to illustrate the improvements in performance over the last few years for PC cluster systems.

Figure 8-a shows the performance of my logarithmic FPGA implementation of the Dirac operator, compared with the performance of a single Intel Pentium 4 3.2 GHz processor from the Pion machine, and a single Intel Xeon 1.7 GHz from the DESY system; both PC processors are running as part of a small cluster of 16 nodes. It can be seen that the performance of the FPGA solution rivals that of the Pion machine's processors, and is several times better than the performance of the DESY machine's processors.

It also can be seen that PC clusters have seen dramatic improvements in performance over the past few years. These improvements have derived from both improvements to the hardware and from improved use of this hardware. Processors, memory buses and communications systems have all increased dramatically in speed and this has had a significant effect on performance. Also users of PC clusters are using the cache pre-fetch instructions to pre-fetch operand data into the processors cache, minimising delays due to memory access. In light of this pace of technological change the performance of my FPGA solution compares well.

8.4 Double precision implementation results

Table 8-c shows the performance for the Dirac operator implemented using IEEE double precision, along with the results for the IEEE double precision based conjugate gradient application. The results shown in Table 8-c are for designs placed and routed for a Xilinx Virtex-II XCV6000 speed grade 6 device. The performance for the core Dirac operator is very strong at 1200 MFLOPS and the performance for the conjugate gradient application is similarly good at 940 MFLOPS. The percentage utilisation of the floating point units is very good for both designs and in particular for the Dirac operator. Achieving 78% utilisation of the arithmetic units used in the

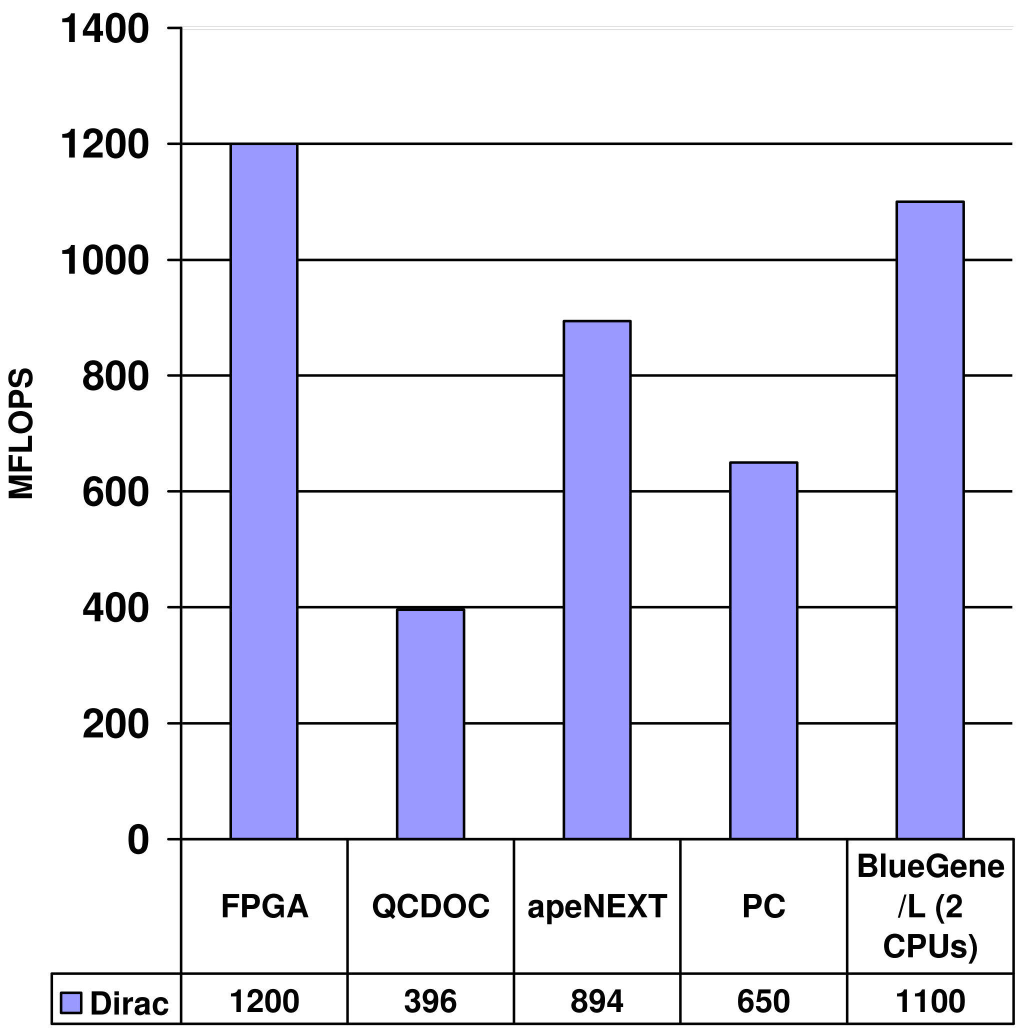
Operator	**Clock Rate (MHz)**	**Performance (MFLOPS)**	**FP Ops per cycle**	**% FP unit utilisation**
Dirac	85	1200	14.1	78.3%
Conjugate Gradient	85	940	11.1	58.4%

Table 8-c. Performance of the double precision FPGA implementations

Dirac operator demonstrates the efficiency of the Dirac operator design.

Many more lattice QCD systems use double precision floating point, than use single precision. Both of the custom ASIC solutions, apeNEXT and QCDOC [Belletti '06][P. A. Boyle '05], use double precision floating point, as do commercial supercomputers such as IBM's BlueGene [A. Gara '05]. Performance figures for the BlueGene system were published by Bhanot et al [Bhanot '05]. The performance figures for the BlueGene system are for two CPUs. Each processing node in a BlueGene machine consists of a dual-core ASIC chip which has two CPUs. The two CPUs work co-operatively on the same problem. Consequently, performance is shown for the combined performance of the two CPUs (see section 3.2.1 for details). PC clusters can also used for double precision lattice QCD simulations and I compare the FPGA version to double precision results published by Holmgren [Holmgren '05a].

Performance of PC clusters is generally halved at double precision when compared with single precision, since memory bandwidth requirements are doubled for double precision. Consequently PC clusters are not as important for double precision simulations as they are for single precision simulations. Commercial supercomputers such as IBM's BlueGene return excellent performance for lattice QCD, but are quite expensive when compared with the custom ASIC systems, so it is relatively rare that a commercial supercomputer is used solely for lattice QCD.

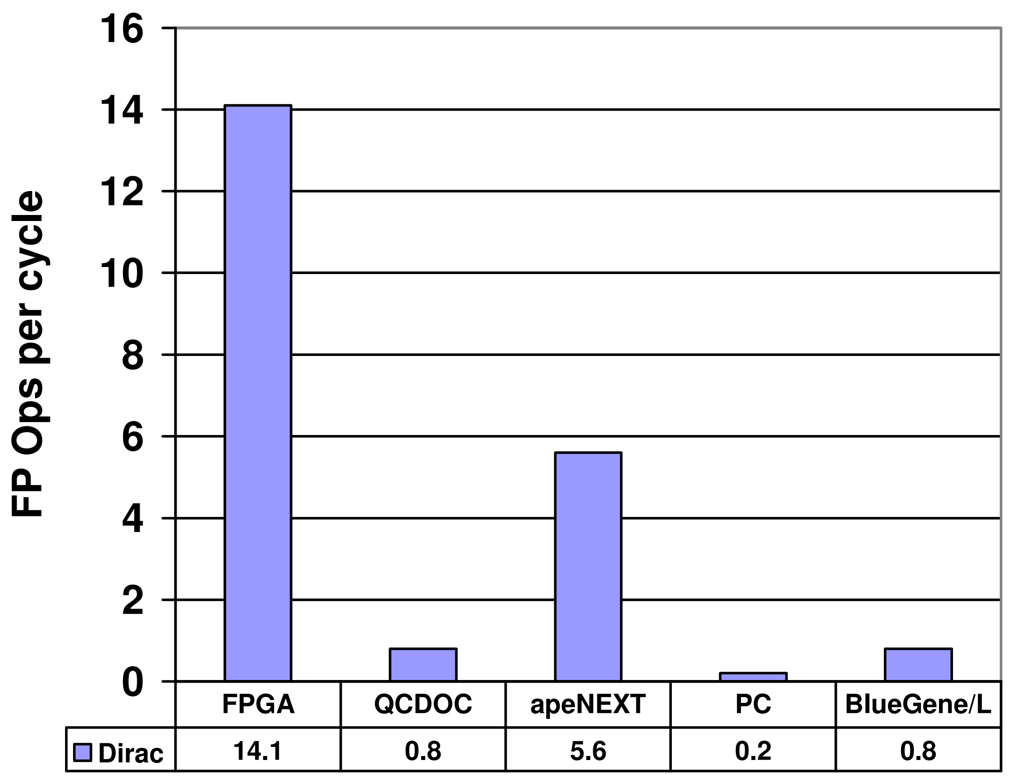


	FPGA	QCDOC	apeNEXT	PC	BlueGene /L (2 CPUs)
▪ Dirac	1200	396	894	650	1100

Figure 8-b. Performance of double precision Dirac operator and comparable systems

The purpose of both custom ASIC machines is to provide as many MFLOPS for as few euro (or dollars) as possible. Both can be used to construct systems of thousands of processors, all of which can work on a single problem. The systems use highly customised low latency interconnection technology to minimise communications delay. Also both systems allow the communication of data between processors to be parallelised with calculation which minimises performance loss in multiple processors systems.

The apeNEXT systems takes the customisation a step further by implementing the *complex-number multiply-accumulate* operation in hardware. This operation is the

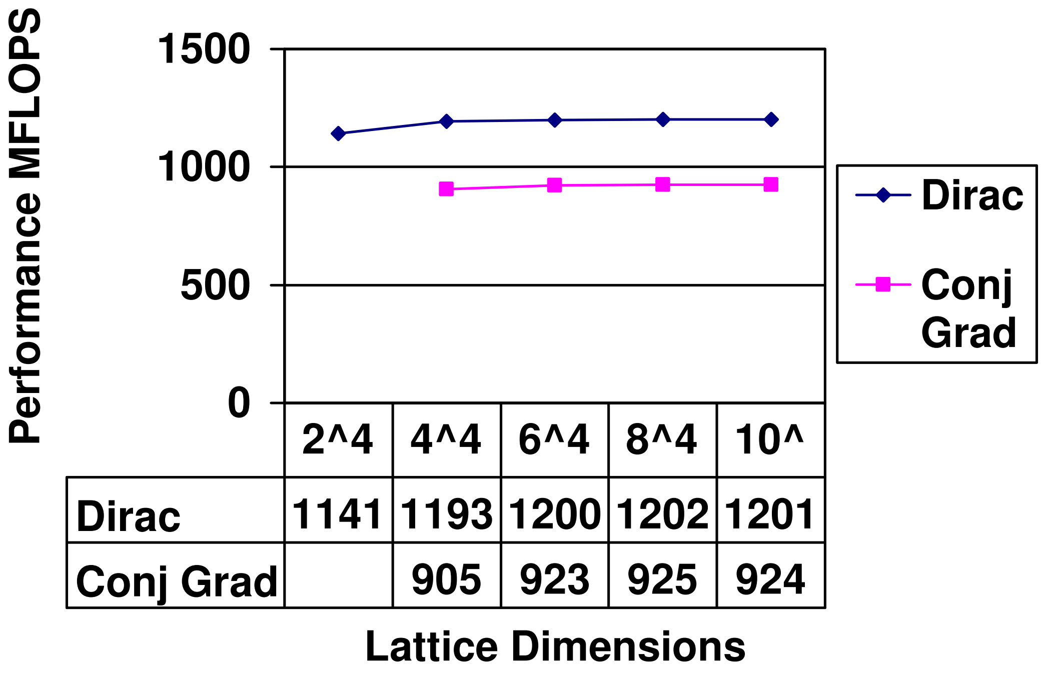


	FPGA	QCDOC	apeNEXT	PC	BlueGene/L
■ Dirac	14.1	0.8	5.6	0.2	0.8

Figure 8-c. Floating Point operations performed per cycle by various lattice QCD systems

most common operation in lattice QCD codes and by implementing it directly in hardware the apeNEXT system obtains excellent performance of nearly 900 MFLOPS per processor from a clock rate of only 160 MHz. The QCDOC processor consists of an IBM PowerPC processor combined with a floating point arithmetic unit running at 460 MHz, which returns performance of nearly 400 MFLOPS per processor. Consequently the QCDOC system relies on using many more processors on a single problem than the apeNEXT system, and to this end it has a more highly optimised communications system.

Figure 8-b shows the performance of my double precision FPGA Dirac operator compared with the performance of several other competing systems. It can be seen that the performance of the FPGA implementation is significantly higher than any of the other solutions shown. The performance of the FPGA version is particularly impressive when compared with that of the PC processor, a 3.2 GHz Pentium 4. The FPGA design has by far the lowest clock rate of any of the systems shown at 85



	2^4	4^4	6^4	8^4	10^
Dirac	1141	1193	1200	1202	1201
Conj Grad		905	923	925	924

Figure 8-d. Effect on performance of increasing problem size for the double precision FPGA designs. Performance is constant for problem sizes larger than 6^4.

MHz. The other systems clock rates are: 500MHz for QCDOC, 160 MHz for apeNEXT, 3.2 GHz for the PC and 700 MHz for BlueGene/L. Figure 8-c shows the number of floating point operations per cycle that each system can sustain for a lattice QCD simulation. This chart shows the extent to which parallelism is exploited in the FPGA designs.

8.4.1 Effect of problem size on double precision FPGA performance

Figure 8-d shows the effect on performance of increasing the problem size for the double precision Dirac operator and conjugate gradient solver FPGA designs. It can be seen that the performance of both designs is constant for all problem sizes above 6^4 and is almost constant for problem sizes above 4^4. The smallest problem size of 2^4 shows slightly lower performance than the larger problem sizes since the efficiency of the Dirac operator pipeline is lowered by the small problem size. The Dirac operator pipeline must be filled and flushed on each use and when the problem size is small these fills and flushes have a noticeable effect on performance. The effect of the fills and flushes diminishes with increasing problem size as the pipeline is full for more of the calculation time. These results demonstrate that by pre-fetching data into temporary storage on the FPGA, memory operations and computation can be parallelised, maintaining maximum performance with increasing problem size.

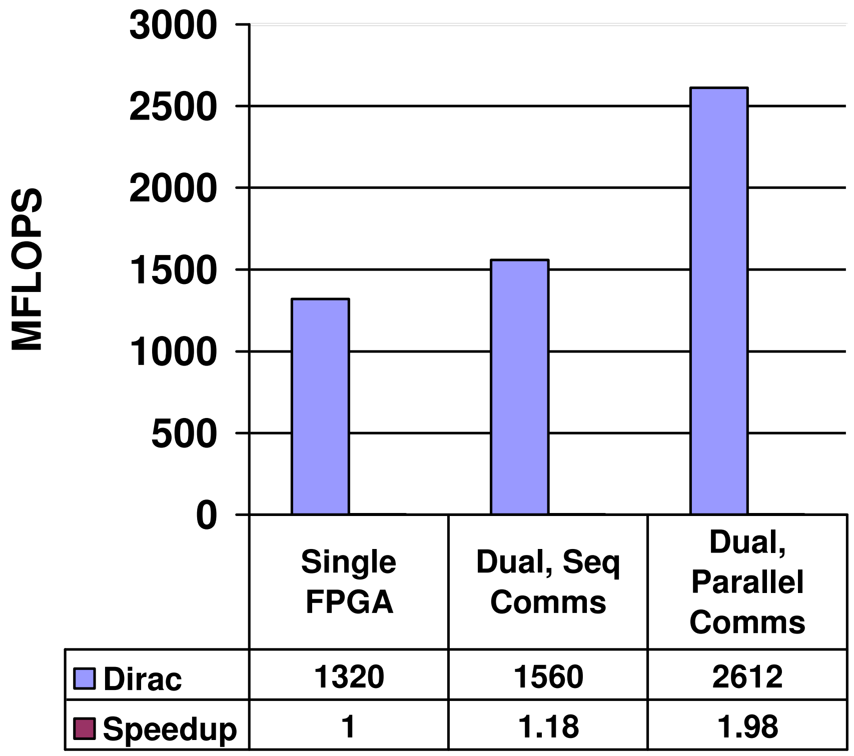
The 24MB of memory available on the FPGA board used to test the designs limited the maximum problem size to 10^4. If a board with more memory had been available then the performance for larger problem sizes could have been easily tested. No results for the conjugate gradient solver running at a problem size of 2^4 are presented since the run time of the solver at this problem size was too short to measure accurately. Large problem sizes require more iterations to find a solution using the conjugate gradient solver. For a problem size of 2^4, in the order of ten iterations are needed to find a solution, and the small problem size means each iteration takes very little time. Consequently it was not possible to accurately measure the performance of the conjugate gradient solver for a problem size of 2^4.

8.5 Dual FPGA implementation results

8.5.1 Measuring performance for the dual FPGA Dirac operator

To run the dual-FPGA design I modified the original host program used for the single-FPGA Dirac operator to run the dual-FPGA version. In order to ensure correct operation of the communications the host program first downloads the appropriate bit file to each FPGA, and then once the download is complete the receivers on each FPGA are started. Once the receivers are up and running both transmitters are started. The communications system does not work correctly if it is not started up in this order since the transmitter does not test to see if the receiver is ready to receive data.

The communications system is not completely reliable, as described in section 7.8. However it is sufficiently reliable to allow the dual-FPGA Dirac operator to run for several seconds without error. This is a sufficiently long period of time to collect valid performance results for the dual-FPGA system. Results are collected in the same fashion as for the single node versions by using the host computers system clock to measure the run time of the FPGA design for several thousand iterations.



	Single FPGA	Dual, Seq Comms	Dual, Parallel Comms
🟦 Dirac	1320	1560	2612
🟥 Speedup	1	1.18	1.98

Figure 8-e. Performance of single & dual FPGA Dirac operators

8.5.2 Performance Results

Figure 8-e shows the performance of the dual FPGA Dirac operator compared with the performance of a single FPGA. Performance is shown for the dual-FPGA with parallelised communications (where result data is transmitted during calculation), and with sequential communications (where calculation waits while the results are communicated).

The sequential dual-FPGA version shows only a small performance improvement of 18% over the single FPGA; in this version the arithmetic units are idle for a large portion of the time whilst the results of an iteration of the Dirac operator are communicated between the FPGAs. By parallelising the communication of data with the calculation, as detailed in section Chapter 7, the performance of the dual-FPGA version is almost double that of a single FPGA. Synchronisation of the two FPGAs

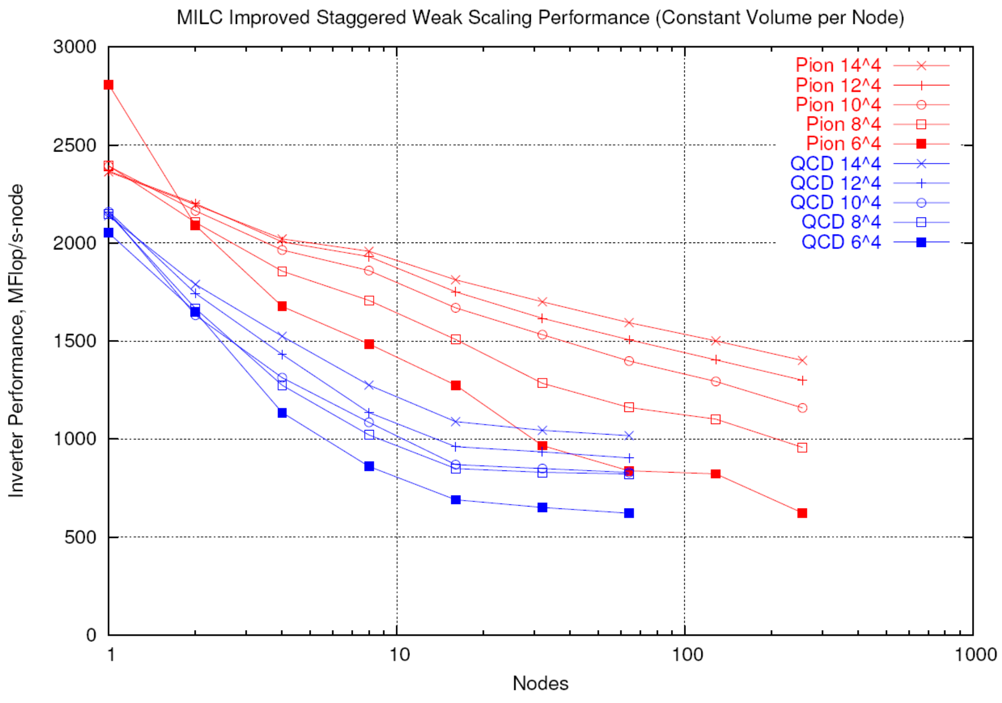


Figure 8-f. Scaling properties of lattice QCD targeted PC clusters. Performance is for single precision arithmetic. Reproduced from [Holmgren '06]

causes the slight reduction; both FPGAs must stop running any calculation during a synchronisation causing the reduction in performance. However since the synchronisation mechanism is very efficient the slowdown is small. The synchronisation mechanism is essential for the dual FPGA Dirac operator to be used for a larger application. In such an application the result of an iteration of the Dirac operator will be used by another operation. Thus the Dirac operator is not run continuously; it must be possible to control its use so both FPGAs work on the same iteration.

8.5.3 Scalability Comparisons

Having demonstrated that the dual FPGA shows excellent scalability, it is useful to show how its scalability compares to that of other lattice QCD systems, and it is useful to consider whether this scalability could be continued for a larger number of FPGAs. Figure 8-f, reproduced from [Holmgren '06], shows the scaling properties of two PC clusters dedicated to lattice QCD at the Fermi National Accelerator

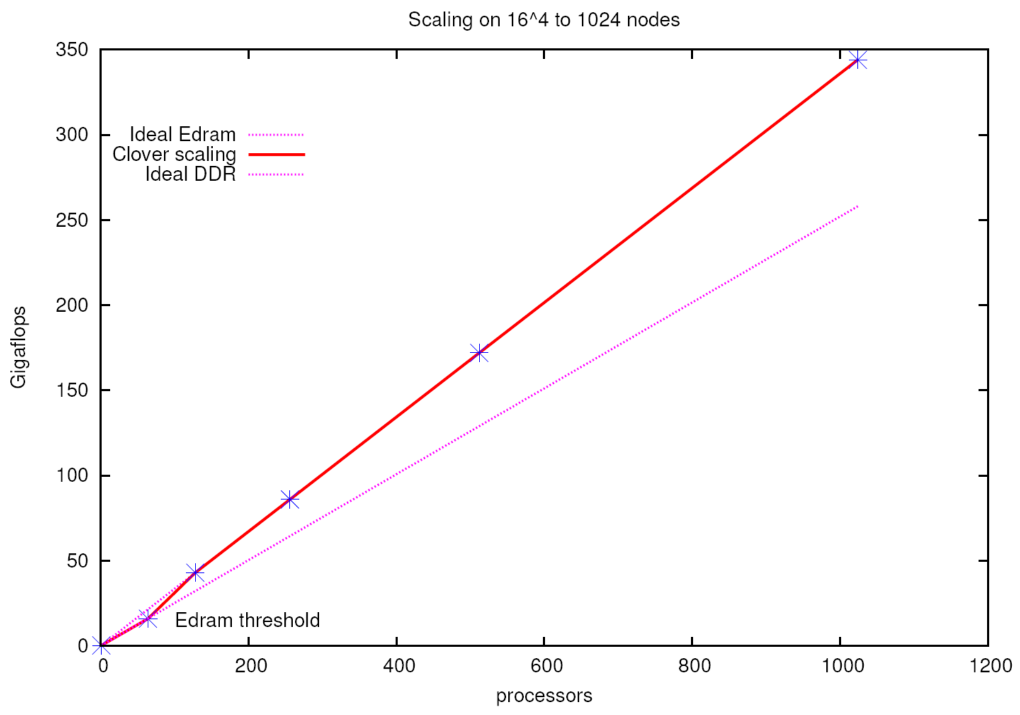
Laboratory, or Fermilab, in the USA. Performance figures are for single precision arithmetic. The Pion machine is the machine used for comparison in both the log arithmetic and double precision results sections of this thesis. The QCD system is slightly older than Pion, with slightly lower performance, and it uses Myrinet for interconnect, whereas the Pion system uses Infiniband.

We can see that both systems have excellent performance for a single node; however performance is degraded badly when more than one node is used. In the case of the QCD cluster, per-node performance using two nodes is only 65% of the performance of a single node for a problem size of 10^4 sites per node. This degradation continues as more nodes are used on a single problem space; the per-node performance of a 16 node QCD system is only 36% of the single-node performance. This is the critical problem with using PC clusters for lattice QCD; the performance for a single node is excellent, but much of this performance disappears when PCs are used in a cluster. The priority, therefore, for designers of lattice QCD machines is to improve the scalability of the systems so that more of the single node performance is preserved when the PC processors are used in a cluster.

The results in Figure 8-f for the Pion system show that improvements in interconnect technology are improving the scalability of PC clusters. The nodes in the Pion system retain 66% of the performance of a single node for a 16 node cluster with a problem size of 10^4 sites per node, compared with only 36% for the same conditions on the QCD machine. Consequently the scalability of PC clusters for lattice QCD is improving, but high speed interconnects are expensive and make up a significant portion of the total system cost, reducing the cost effectiveness of systems that use these interconnects. Gigabit Ethernet is a lower cost alternative to Infiniband or Myrinet, however it suffers from much high latencies which restricts the scalability of systems that use Gigabit Ethernet for their interconnect. Building PC clusters for lattice QCD is a complex process with a lot of variables that affect the performance; a lot of care is needed to choose the best components.

Low latency is the key to scalability for lattice QCD systems. Infiniband has lower latency than Myrinet; hence the Pion system shows better scaling than the QCD

Figure 8-g. Scalability of the QCDOC machine



system. Infiniband has a typical latency of about 3.5µs, whereas Myrinet's latency is about 5 µs and Gigabit Ethernet is about 12 µs, [Wettig '05]. The most scalable systems for lattice QCD are the custom ASIC systems, apeNEXT and QCDOC. Communications latency for both of these systems is about 0.5 µs which is very low and is partly responsible for the scalability of these systems. Both systems also allow communications to be “hidden” by performing communication in parallel with computation, which, when combined with the low latency communications, gives excellent scalability for both of these systems.

Both apeNEXT and QCDOC can sustain excellent per node performance on systems consisting of thousands of nodes. Figure 8-g, reproduced from [Wettig '05] shows the total system performance for a fixed total problem size and a variable number of nodes for the QCDOC system. There is a linear relationship between the number of nodes used and the total system performance, which shows that the QCDOC system has excellent scalability for large systems.

The dual-FPGA communications system has very low latency, and allows parallelisation of communication and computation. These two factors are the key to the scalability of the custom ASIC systems. Latency for the dual-FPGA system is less than 0.25 μs for small message sizes, which easily matches that of the custom ASIC machines. This low latency, when combined with the parallelisation of computation and communication for the Dirac operator strongly suggests that the FPGA system would scale to a significant size.

8.6 Notes on comparing performance of lattice QCD machines

In the previous sections the FPGA designs were compared with performance data for three classes of computing machines; PC-based clusters, commercial supercomputers and custom ASIC based supercomputers. The PC clusters and the custom ASIC supercomputers are built specifically for lattice QCD calculations. All the data used for comparison is taken from the literature. The performance figures for these three classes of machine are heavily influenced by two factors; memory access and inter-processor communication performance.

The performance data for the commercial supercomputers and for the custom ASIC supercomputers is limited in detail. Generally performance data for these machines is only published for one processor running as part of a multiple processor machine. These machines have the ability to pre-fetch data into on-chip cache memory before it is required. Thus the performance of these processors is not sensitive to cache effects. Combined with the predictable access patterns of lattice QCD applications, this means that changing the per-processor problem size has relatively little effect on the processor's performance.

The performance of multi-processor machines used for lattice QCD is sensitive to the latency of the inter-processor communication system; this is discussed in detail in 8.5.3. The supercomputer machines also have very low latency communication systems; as a result communication delay has little effect on per processor performance for these machines. The custom ASIC supercomputers can further improve performance by parallelising communication and calculation which helps prevent communication delays from degrading performance, Figure 8-g shows that total performance for the QCDOC machine is a linear function of the number of processors used in the machine, which demonstrates the excellent scalability of this class of machines.

The situation for the performance of lattice QCD on PC clusters is quite different. PC processors show multi-GFLOP single precision performance for a small problem. However much of this performance is lost if multiple processors are used on a single problem. The communication systems used for PC clusters have improved significantly in recent years, however the effect of communications latency is still significant compared with commercial supercomputers. Also it is still not possible to effectively parallelise computation and communication on PC clusters. Consequently PC clusters show sub-linear scalability, with per-processor performance degrading steadily with increasing cluster size. At the time of writing no lattice QCD performance results for multi-core PC processors is available. Thus it is only possible to compare the FPGA designs with single core PC processors.

Figure 8-f shows how the per-processor performance of two PC clusters falls as the number of processors used on a single problem is increased. The Pion system, which is one of the most advanced PC clusters used for lattice QCD, suffers a 35% loss in per-processor performance when the number of processors used is increased from one to sixteen for a moderate per-processor problem size of 8^4. For a large system of 256 processors the per-processor performance drops by 60%, compared with the single processor system.

It is impossible to generate a significant scientific result for lattice QCD with a single processor; the problems are simply too large. Consequently the high performance of a single PC processor system is not very relevant to lattice QCD. It is far more relevant to discuss the performance of PC processors when used in a small cluster, since this is the environment in which they will actually be used. Also since performance data for other solutions is generally quoted for a single processor running in a multiple processor system, it is the most valid metric for comparison. Only quoting PC performance for a single processor machine would give the misleading impression that PC clusters are far better than any other machine for lattice QCD, when in fact they not, owing to the significant loss of per-processor performance when used within a large cluster.

It was not possible to investigate the performance of an FPGA based lattice QCD system consisting of many FPGAs, since suitable equipment was not available. However the per-FPGA performance of the dual FPGA system is 99% of the performance of the performance of the single FPGA. In comparison the performance of the latest PC cluster, shown in Figure 8-f, drops by between 12.5% and 25% when two processors are used instead of one. This indicates that an FPGA based system would have significantly better scalability than a PC cluster based system.

For large systems the communications pattern for the Dirac operator is a nearest neighbour pattern where each processor only communicates with its nearest neighbour in the system. The only operator that does not have a nearest neighbour communications pattern is the global sum or dot-product operator, where each processor sums its portion of data and the results of each processors sum must be summed together. A low latency network is the key to performing this operation efficiently.

The FPGA based system has very low latency communications, and it is capable of completely parallelising communication and computation. These characteristics are the key to the linear scaling properties of both the QCDOC and apeNEXT custom ASIC supercomputers, which suggests that a system using many FPGAs would potentially have similar scaling characteristics to these custom ASIC supercomputers.

8.7 Summary

This chapter has presented results for both log arithmetic and double precision floating point based FPGA implementations of the Dirac operator and a conjugate gradient solver. The per-processor performance of the log arithmetic designs compares well to that of PC cluster based machines at 1320 MFLOPS for the FPGA, compared with the latest PC cluster at 1300 MFLOPS. The performance of the double precision designs compares even more favourably with alternative machines, at 1200 MFLOPS for the FPGA compared with 894 MFLOPS for the apeNEXT custom ASIC supercomputer, 650 MFLOPS for the latest PC cluster and 396 MFLOPS for the QCDOC custom ASIC supercomputer. This result demonstrates that FPGAs can return competitive performance for a typical high performance computing application, using IEEE double precision arithmetic.

The results for the dual-FPGA Dirac operator based on the log arithmetic Dirac operator demonstrate that two FPGAs can be used effectively together on a single lattice QCD problem. The low latency interconnects used for this design allows the dual FPGA design to deliver performance that is 1.98 times better than a single FPGA. Furthermore an analysis of the latency of the communications system used for the dual FPGA Dirac operator shows that it has a latency that is lower than highly scalable systems such as QCDOC or apeNEXT. The performance of the dual FPGA Dirac operator, combined with the analysis of communications latencies of lattice QCD systems, shows that FPGAs have the potential to make a scalable multiple processor platform for lattice QCD.

Chapter 9

Final Thoughts

In this thesis I have presented a number of designs that perform lattice QCD calculations using FPGAs. The thesis describes these designs and presents performance results for these designs running lattice QCD simulations. These performance results are compared with a number of highly optimised state of the art computing machines that are used for lattice QCD simulations. This chapter presents final reflections on the results presented and draws conclusions from the work.

9.1 The Suitability of FPGAs for High Performance Computing

In recent years large FPGAs and the availability of non-integer arithmetic cores have made FPGAs a potentially viable platform for high-performance scientific computing applications. Research has shown that the peak performance of FPGAs has grown very quickly in recent years [Underwood '04a]. A single FPGA can now support a significant number of non-integer arithmetic units, which when used in parallel can deliver very high peak performance. However exploiting this fine grain parallelism is difficult for real applications.

There is a substantial body of research that is concerned with FPGA implementations of common kernels such as dense matrix multiplication, dot product operators, and matrix by vector multiplication [Dou '05][Underwood '04b][Zhuo '04]. This research has shown that FPGAs can return excellent performance for these kernels and this result is very valuable.

Despite this, there has been little study of how well FPGAs perform for real complete scientific computing applications. The existing research on important kernel operations does not examine how these kernels can be used to implement full scientific computing applications. Much of this research does not consider memory bandwidth constraints for realistic FPGA platforms, and it does not consider whether multiple on-FPGA processing units can be usefully exploited for real applications.

9.2 Contributions of this Thesis

Lattice QCD is an important scientific application and is the focus of considerable research work worldwide. A substantial amount of research effort has been expended on producing computing machinery for lattice QCD including two competing custom ASIC based supercomputers and a variety of specially designed PC clusters. Commercial supercomputers, such as the IBM BlueGene/L machine, are also used for lattice QCD. This makes lattice QCD an excellent application for evaluating a computing platform's suitability for scientific computing.

I have presented the design and implementation, for FPGAs, of the core Dirac operator and a full lattice QCD application using IEEE double-precision floating-point. I have also presented a version of the log arithmetic Dirac operator that uses two FPGAs in parallel on a single problem.

As discussed in Section 3.2.2, either single or double precision can be used for lattice QCD, however double precision is preferred because it is more accurate. If a system has significantly higher performance for single precision compared with double, then single precision will be used on that system. This is the case for PC clusters where double precision performance is usually half that of single precision, however all of the other systems use double precision arithmetic. Consequently the IEEE double precision implementations are compared with all the other solutions but the LNS implementation is only compared with the performance of PC clusters. Floating point operations per second (FLOPS) is used as the metric for comparison; this is the standard measure for lattice QCD machines.

Log arithmetic cores were used in this work because floating point cores were not available at the start of the project. The log arithmetic designs perform well, achieving **1320** MFLOPS for the Dirac operator and **1050** MFLOPS for the full conjugate gradient application. This compares well with the *single precision* performance of a PC cluster node of 1100 MFLOPS for the Dirac operator. Nonetheless, Lattice QCD, like most scientific applications that operate on matrices, has roughly the same numbers of additions and multiplications, and it has few divisions. The large block RAM tables required by LNS adders were always the limiting factor in the log arithmetic design.

IEEE format floating point units have no such limitations, so it was possible to build IEEE *double precision* floating point implementations that achieve **1200** MFLOPS for the Dirac operator and **940** MFLOPS for the full application using ten double precision adders and eight multipliers. The double precision implementations are far more complex, however, because fewer multipliers are available compared with the logarithmic arithmetic designs and because the multiplier pipelines are deeper. For the double precision designs memory bandwidth, available block RAMs and available slices are all critical constraints on the performance of the designs.

The per processor performance results for the double precision implementations compare extremely well with both the ASIC solutions, and also with the performance of a PC cluster at double precision. QCDOC prototypes return 535 MFLOPS per node whilst apeNEXT nodes return about 896 MFLOPS per node. PC cluster nodes return about 550 MFLOPS at double precision for the Pion system described by Holmgren [Holmgren '05a].

The double precision implementations also compare well with the per node performance of commercial supercomputers. The IBM BlueGene/L returns performance of 1100 MFLOPS per processing node. Each processing node consists of two PowerPC CPU cores, so the average per core performance of the BlueGene is 550 MFLOPS. Once again the performance of the FPGA implementations compares well. All figures are for the performance critical Dirac operator.

The results show that FPGAs can be competitive with general purpose processors and even custom ASIC processors for scientific computing applications such as lattice QCD. To my knowledge this is the first FPGA implementation of lattice QCD and one of the first full implementations of a large scientific application using IEEE double precision arithmetic.

A single FPGA will never be able to meet the performance requirements of lattice QCD. All the machines currently used for lattice QCD calculations are massively parallel machines that use many processors on a single problem. An FPGA based machine for lattice QCD, or for any significant scientific computing application, would be no different. Consequently I developed an inter-FPGA communications system and used it to create a version of the logarithmic arithmetic Dirac operator which runs on two FPGAs.

This dual-FPGA version of the Dirac operator shows a speed up of 1.98 times over the performance of the single FPGA version of the Dirac operator. Data communication for the dual FPGA version is completely parallelised with computation so computation never has to stop whilst waiting for communication to complete. This result demonstrates that FPGAs have the potential to be used to create scalable multiple FPGA machines for scientific computing.

The communications system used for the dual FPGA Dirac operator has a lower latency (0.25 µs) than the communications systems used in highly scalable systems such as QCDOC or apeNEXT (0.5 µs). The FPGA communications system's latency is substantially lower than that of Infiniband, which is the best interconnect available for PC clusters, at (3.5 µs). The performance of the dual FPGA Dirac operator, combined with the analysis of communications latencies of lattice QCD systems, shows that FPGAs have the potential to make a scalable multiple processor platform for lattice QCD.

9.3 Limitations of This Work and Suggestions for Future Work

The intention of this thesis is to investigate the technical feasibility of using FPGAs for scientific computing applications, and it has been shown that FPGAs can return excellent performance for a real scientific computing application. However FPGAs are very expensive when compared with commodity CPUs like those used in PC cluster machines. It is difficult to obtain exact pricing for FPGAs but the Xilinx Virtex-II FPGA used here costs several times the price of a good PC processor. Smaller cheaper FPGAs are available, but these have significantly lower performance. Thus, currently, cost is a significant barrier to real-world use of FPGAs for scientific computing. However new generations of FPGAs may change this cost relationship by delivering more performance for the money.

The FPGA used in this project is now somewhat out of date. Two successive generations of FPGAs have been released since the release of the FPGA used here; the Virtex-4 and Virtex-5 families. Each of these generations has delivered very substantial improvements in clock rate over the previous generation, which would translate to considerably improved performance for the designs described in this thesis. Also these new FPGAs have different combinations of conventional FPGA slice logic, hardware multipliers and block RAMs, which could improve the performance of non-integer arithmetic on these new FPGAs. However the performance of these FPGAs for scientific computing applications has not yet been analysed and this would be an area worthy of future work.

Recent publications have made predictions for the future floating point performance of FPGAs and commodity CPUs. The predictions (based on recent trends) predict that the peak and sustained performance of FPGAs will soon be superior to that of commodity desktop processors [Underwood '04b]. These predictions are now less certain since they do not account for the latest, very significant, innovations in both FPGAs and commodity processors. These predictions do not consider the Virtex-4 or the Virtex-5 FPGAs nor do they consider the latest developments in multi-core commodity processors.

Multi-core processors have significant potential for scientific computing. Multi-core processors consist of multiple processors integrated onto a single processor die, often sharing memory bandwidth and caches. Traditionally, the performance of commodity CPUs has been improved by raising the clock rate at which the chips run. However recently it has become very difficult for processor designers to continue to raise the clock speeds of their processors. Multi-core processors were conceived as a way of improving the performance of commodity CPUs, without having to raise clock rates to very high levels. This provides a low-cost and power efficient way of improving the processing power of commodity processors.

The cores in multi-core processors share memory bandwidth, which could cripple their performance for many applications. However the multiple on-chip cores share cache facilities which, if data is handled carefully to maximise reuse between the two cores, could allow these multi-core processors to return excellent performance for scientific computing applications. Thus an interesting area of future work would be to compare the performance of multi-core commodity processors with that of the latest generations of FPGAs, to determine how both platforms compare for scientific computing.

Another limitation of this thesis is the approach taken to implementing the FPGA designs described here. In order to achieve maximum possible performance for the FPGA designs, I decided to implement highly specialised designs that are tailored to the precise requirements of lattice QCD. This is an acceptable approach for lattice QCD since the performance critical code components are well established and do not change. However the disadvantage of this approach is that a substantial redesign is required in order to implement a different algorithm. A hybrid design consisting of a processor running software code, attached to dedicated hardware running on an FPGA would provide a more flexible approach. The processor could be part of the FPGA fabric, as in the Virtex-II Pro FPGA, or it could be a conventional processor which is closely coupled to the FPGA. This approach would likely return poorer performance than dedicated hardware for a particular application, however the flexibility and shorter design times of the system could compensate for this.

9.4 Conclusion

This thesis builds on the existing work on common scientific computing kernels to show that FPGAs can be a viable platform for real scientific computing applications. Designs have been implemented that show FPGAs can return performance that is competitive with state of the art computing machinery for a real and significant high performance computing application. Also the dual FPGA implementation of the

Dirac operator demonstrates that the customisable nature of FPGAs allows inter-FPGA communication to be nearly completely parallelised with computation. This allows the dual-FPGA design to run at nearly twice the speed of the single FPGA version. A comparison of the latencies of the FPGA communication system compared with that of various alternative lattice QCD systems shows that the FPGA system has the lowest latency, father supporting FPGAs as a viable, scalable, high performance computing platform. These results show that FPGAs have considerable potential as a platform for high performance computing applications.

Appendix A

Pipelined Use of Arithmetic Units

This appendix describes how the logarithmic arithmetic pipelines are used in the logarithmic Dirac operator. This appendix is not required to understand the logarithmic Dirac operator; it is included to complement the design description contained in Chapter 5. The lattice QCD algorithms implemented in this work were supplied as C source code. To create the highly optimised designs described in this thesis, I first created a very basic FPGA implementation of the Dirac operator. This operator exploited very little parallelism, and consequently had poor performance. However it formed the basis for later versions of the Dirac operator. The final versions of logarithmic operator were created by incrementally improving this basic Dirac operator. One of the most significant improvements was to make pipelined use of the arithmetic units assigned to each part of the algorithm. The assignments are described in section 5.1.2.

A.1 Gamma Functions

The code in Example A-i shows the structure of a basic version of a gamma operation. Each call to ladd1 or lsub1 takes 10 cycles to complete, there is considerable scope for performance improvement through pipelining use of the adder arithmetic unit. The Flip function is a zero cycle function which inverts the sign of the number passed to it, checking first that the number is not zero. Unlike IEEE floating point the LNS system used in this project has no negative zero thus if the sign of zero is flipped then the number becomes a Not a Number exception. This is prevented by only flipping the sign of numbers that are not zero.

All the 8 gamma functions are identical in structure. They all have a loop with three iterations and each iteration consists of four additions or subtractions followed by a set of assignments of the results of those operations to other points in the results matrix. The locations in the matrix that are operated on, and whether they are added or subtracted, differ with different versions of the function but the structure remains the same. This means that optimisations that work for one version of the function, will work equally well for all versions.

Example A-i. Gamma function before pipelining

```
void HG5pG5Gy(l_real (*rr)[3], l_real (*ir)[3], l_real
(*ra)[3], l_real (*ia)[3])
{
  unsigned c;
  c=0;
  while(c<3){
   rr[0][c] = ladd1(ra[0][c] , ia[3][c]);
   ir[0][c] = lsub1(ia[0][c] , ra[3][c]);
   rr[1][c] = lsub1(ra[1][c] , ia[2][c]);
   ir[1][c] = ladd1(ia[1][c] , ra[2][c]);

   par{
    rr[2][c] = Flip[0](ir[1][c]);
    ir[2][c] = rr[1][c];
    rr[3][c] = ir[0][c];
    ir[3][c] = Flip[1](rr[0][c]);
    seq{
    c++;
   }
  }
 }
}
```

In Example A-i the issuing and retrieval of operands to and from the adder pipes is encapsulated within a 10 cycle function called ladd1 (or lsub1). To improve adder performance, issue of operands and retrieval of results was separated to allow the 12 operands to be issued on 12 sequential cycles, instead of the issues being separated by 9 cycles. This was done by breaking the single loop into two loops; one to issue the operands and the other to retrieve and store the results. The retrieval loop stores each result in two locations in the result matrix. The sign bit of some of the results needs to be flipped. This is performed by the Flip macro, seen in the code example. The converted code is shown in Example A-ii.

Example A-ii - Pipelined Gamma Function

```
void HG5pG5Gy(l_real (*rr)[3], l_real (*ir)[3], l_real
(*ra)[3], l_real (*ia)[3])
{
 unsigned int 4 c, d;
 unsigned 5 cycles;
 signal l_real res;

 /* Issue the 12 add/subs to a single pipe
  This will take twelve cycles
  Need to delay for 9 cycles before collecting first result
  And continue for 12 cycles
 */
 par{
  c=0;
  d=0;
  cycles = 0;
 }

 while(cycles<21){
 par{
  cycles++;
   if(cycles < 12){
    par{
     if(c<-2 == 0){
      la1(ra[0][c[3:2]] , ia[3][c[3:2]]);
     } else if(c<-2 == 1){
      ls1(ia[0][c[3:2]] , ra[3][c[3:2]]);
     } else if(c<-2 == 2){
      ls1(ra[1][c[3:2]] , ia[2][c[3:2]]);
     } else if (c<-2 == 3){
      la1(ia[1][c[3:2]] , ra[2][c[3:2]]);
     } else {
      delay;
     }
    c++;
    }
   } else {
    delay;
   }

   //Now in 9th cycle, first pair of results are ready
   //for collection
   if(cycles > 8){
    if (d<-2 == 0){
     par{
      //Assign reults to a signal; it
      //will hold the value for this
      //clock cycle
      res = lret1();
      //Store results in return array
      rr[0][d[3:2]] = res;
      //Flip returned results
```

```
        ir[3][d[3:2]] = Flip[0](res);
        d++;
       }
      } else if (d<-2 == 1) {
       par{
        res = lret1();
        ir[0][d[3:2]] = res;
        rr[3][d[3:2]] = res;
        d++;

      } else if (d<-2 == 2) {
       par{
        res = lret1();
        rr[1][d[3:2]] = res;
        ir[2][d[3:2]] = res;
        d++;
       }
      } else if (d<-2 == 3) {
       par{
        res = lret1();
        ir[1][d[3:2]] = res;
        rr[2][d[3:2]] = Flip[0](res);
        d++;
       }
      } else {
       delay;
      }
     } else {
      delay;
     }
    }
   }
   return;
  }
```

A.2 *Multiply Functions*

The code contained in Example A-iii shows one of the two multiply functions before arithmetic unit use was pipelined. The function uses two multiplier units and one adder unit to perform its calculation. The operand matrices *ra* and *ia* are the results of one of the gamma functions.

Example A-iii - Non-Pipelined Multiply Function

```
void HMulGl3Wfv(l_real (*rr)[3], l_real (*ir)[3],
                   l_real (*rg)[3], l_real (*ig)[3],
                   l_real (*ra)[3], l_real (*ia)[3])
{
  unsigned 2 c, z;
  unsigned 3 d;
  l_real rtemp, itemp, zero, flag;
  l_real rtemp1, itemp1, rtemp2, itemp2;

  for (d = 0; d < 4; d++){
   for (c = 0; c < 3; c++){
       par{
             rr[d<-2][c] = ZERO;
             ir[d<-2][c] = ZERO;
       }

       for(z =0; z<3; z++) {
         //Part One
         par{
               rtemp1 = lm[0](rg[c][z], ra[d<-2][z]);
               rtemp2 = lm[1](ig[c][z], ia[d<-2][z]);
         }
         par{
               itemp1 = lm[0](rg[c][z], ia[d<-2][z]);
               itemp2 = lm[1](ig[c][z], ra[d<-2][z]);
         }

         rtemp = lsub1(rtemp1, rtemp2);
         itemp = ladd1(itemp1, itemp2);

         //Part Two
         rr[d<-2][c] = ladd1(rtemp, rr[d<-2][c]);
         ir[d<-2][c] = ladd1(itemp, ir[d<-2][c]);
      }
   }
  }
}
```

Performing a matrix multiply with only one adder pipe is non-trivial since the results of the three multiplications performed for each component of each point must be added together, leading to dependencies between some of the addition operations. No-ops must be issued to the adder pipelines to manage these dependencies. The no-ops are implicit in the multi-cycle *ladd1* arithmetic functions. However by calculating all 12 points in parallel the no-ops can be eliminated, and replaced with useful calculations.

The calculation for each point is broken into two stages. The first stage multiplies the real and imaginary parts of the data from a pair of points in the two operand matrices and then adds the results of the two multiplications together. The second stage accumulates the result of this addition to the correct point in the result matrix.

The calculations for the points were parallelised by performing all of the first stage additions, before starting the second stage additions. The first stage multiplications are performed using a pair of multipliers and the results from the two multipliers are issued directly to the adder to be added or subtracted. The results of these additions fall into one of two sets of 36, one set is for the real components of the result matrix and the other set is for the imaginary components. Each of these sets contains 12 sub-sets; the three datum from each sub-set must be added together to calculate the result for each component of each point in the result matrix.

These data are added by first setting the values for the result points to zero, and the three data from each sub-set are then accumulated to the result storage. Accumulating the results in this way works well since the size of the Example A-ivresult matrix is greater than the latency of the adder pipes. This means that once the last add of the first round of accumulate additions has been performed the result of the first add of that round is already complete, so the second round can be started immediately. This eliminates all no-op instructions from the adder pipeline schedule. The code for the optimised matrix multiply function is shown in Example A-v.

The results of the first stage are stored in two 36 element distributed RAMs called *rstore* and *istore*. Once all the additions of the first stage have been issued then second stage additions can begin. All of the elements in *rstore* and *istore* are added to the appropriate positions in the result arrays *rr* and *ir*. This pipelining strategy ensures that the adder pipe is busy from the first issue to the last, there are no idle cycles. This pipelining strategy is used for all 8 multiply functions and each one has exclusive use of an adder and so the utilization of eight of the ten adders available is very high.

Example A-v - Pipelined Multiply Function Code

```
void HMulGl3Wfv(l_real (*rr)[3], l_real (*ir)[3],
    l_real (*rg)[3], l_real (*ig)[3],
    l_real (*ra)[3], l_real (*ia)[3])
{
 unsigned 8 cycles;
 unsigned 4 d, e, h;
 unsigned 6 f, g;
 unsigned 2 z;
 l_real istore[36], rstore[36];

 par{
  cycles = 0;
  d=0;
  e=0;
  f=0;
  g=0;
  h=0;
  z=0;
  par(a=0; a<12; a++){
   rr[a[1:0]][a[3:2]] = ZERO;
   ir[a[1:0]][a[3:2]] = ZERO;
  }
 }

 while(cycles < 154){
  par{
   cycles++;
   //Issue multiplies; increment d then z, this makes
   //the final additions easier
   if(cycles < 72){
    par{
     if (cycles[0] == 0){
     par{
      lmstage1[0](0, rg[d[3:2]][z], ra[d[1:0]][z]);
      lmstage1[1](1, ig[d[3:2]][z], ia[d[1:0]][z]);
     }
    } else {
     par{
      lmstage1[0](2, rg[d[3:2]][z], ia[d[1:0]][z]);
      lmstage1[1](3, ig[d[3:2]][z], ra[d[1:0]][z]);
      if(d==11){
       par{
        d=0;
        z++;
       }
      } else {
       d++;
      }
     }
    }
  }
 } else {
  delay;
}

   //Retrieve results of multiplies and issue to adders
   if ((cycles > 0) && (cycles < 73)){
    if(cycles[0] == 1){
     ls1(lmstage2[0](0), lmstage2[1](1));
    } else {
     la1(lmstage2[0](2), lmstage2[1](3));
    }
   } else {
    delay;
   }

   //Retrieve results of first set of adds and store
   if ((cycles > 9) && (cycles <82)){
    if(cycles[0] == 0){
     rstore[f] = lret1();
    } else {
     par{
      istore[f] = lret1();
      f++;
     }
    }
   } else {
    delay;
   }

   if ((cycles > 72) && (cycles < 145)){
    if(cycles[0] == 1){
     la1(rr[e[1:0]][e[3:2]], rstore[g]);
    } else {
     par{
      la1(ir[e[1:0]][e[3:2]], istore[g]);
      g++;
      if(e == 11){
       e=0;
      } else {
       e++;
      }
     }
    }
   } else {
    delay;
   }

   if ((cycles > 81) && (cycles < 154)){
    if(cycles[0] == 0){
      rr[h[1:0]][h[3:2]] = lret1();
     } else {
      par{
       ir[h[1:0]][h[3:2]] = lret1();
       if(h==11){
        h=0;
       } else {
        h++;
       }
      }
     }
    } else {
     delay;
   }
  }
 }
}
```

A.3 Combining Gamma and Multiply Functions

The gamma and multiply functions are used in pairs; there are eight *gamma* operators, and the result of each of these operators is processed by a dedicated *mul* block. The paired *gamma* and *mul* functions share an adder pipeline. When the *gamma* and *mul* functions are separate, the adder pipeline must be flushed by the *gamma* operator before the *mul* operator can be called. Combining the *gamma* and *mul* operators into a single operator, which I will call the *gamma-mul* operator, allows the pipeline flush to be eliminated. By combining the two operators into one single operator, the *mul* section can begin issuing operations to the adder pipeline as soon as the *gamma* operator has finished issuing operations to the pipeline. Each of the *gamma* operators was combined with its paired *mul* block to form eight *gamma-mul* blocks.

Appendix B

Clock Rate Improvement Example

This appendix describes how the clock rate of the logarithmic Dirac operator was improved through repeated place and routes and timing analysis. This appendix shows some common sources of delay in FPGA designs, and it shows how they were eliminated for the logarithmic Dirac operator.

The first step in clock rate improvement is to find the longest delay in the design. Visual inspection of the code can highlight some obvious sources of delay; however it is not sufficient on its own. A much more comprehensive method is required to ensure that clock rate is maximised. The method described in this chapter involves placing and routing the design and then using Xilinx Timing Analyser to find the longest delay in the design. This delay is then traced back to the source code, which can then be altered to reduce the delay.

B.1 Sources of Delay

The single cycle timing model used in Handel-C means that any single line of code must be completed within a single cycle. However if a single line of code contains a very complex operation then it will cause a long delay, resulting in a low clock rate for the whole design. For example the following code could be clocked at a high frequency, since the logical and operator is simple:

```
unsigned 64 a, b, c;
c = a & b;
```

In comparison the code below is complex and must be clocked with a slower clock. Integer multiplication is far more complex than a logical *and,* and is a common source of low clock rates in FPGA designs.

```
unsigned 64 a, b, c;
c = a * b;
```

Condition checks on if statements and while loops are another common source of delay; the condition check must be performed in sequence with the body of the statement or loop within a single clock cycle. Thus the condition check needs to be as simple as possible. The best way to do this is to evaluate the condition on the previous cycle, saving the result in a one bit register. This register is then used to control the loop or if- statement.

Delay can also be introduced in less apparent ways. If a resource is heavily shared throughout a design then multiplexers must be constructed to control access to the resource. Heavy use of a single resource also leads to a lot of long wiring. Thus duplicating resources can often make significant improvements in clock rate by reducing multiplexer delay and by reducing the length of wiring.

B.2 Delay in the Dirac Pipeline Control Structure

The first phase of timing analysis revealed that the longest paths in the design were all related to the control structure used to control access to the pipelined arithmetic units. A complex structure of nested *if* statements and *for* loops was used to control issue of operands to and retrieval of results from the arithmetic units. Nested if statements must be evaluated sequentially, causing a long logic delay. The following code shows how nested if statements were used in the design.

```
while(cycles < 155){
  if(cyclesG < 12){
    if(c == 0)
       // one cycle of code
    else if(c == 1)
       // one cycle of code
    else if(c == 2)
       // one cycle of code
    else if(c == 3)
       // one cycle of code
    else
       delay;
  }
}
```

Example B-i - Complex Control Code

The nested if statements in Example B-i contribute to a long logic delay for this part of the design. Other causes of delay in the example are the conditions on the loop and the first if statement. Example B-ii shows an improved version of the code in Example B-i. The four parts of the nested if-statement are evaluated in parallel and the condition checks of the loop and the first if statement is now evaluated during the previous cycle. These two improvements significantly reduce the delay of the code section shown.

```
while(loopDone == 1){
  if(cycles == 154)
    loopDone = 0;
  if(issueGamma == 1){
    par{
       if(cyclesG == 11)
         issueGamma = 0;

       if(c == 0)
         //one cycle operation
       if(c == 1)
         //one cycle operation
       if(c == 2)
         //one cycle operation
       if(c == 3)
         //one cycle operation
    }
  }
}
```

Example B-ii - Optimized Control Structure

The improved control structure was applied to all eight *gamma-mul* blocks by integrating the eight blocks into a single operator. This reduced the resources required by the control structure and simplified the process of optimizing the control structure.

B.3 *Pipelining the SITE Calculation*

The SITE calculation is used to calculate the memory address of data in off-chip memory given the co-ordinates of a site in the lattice data set. The SITE calculation is shown Example B-iii below.

```
macro proc SITE(x, y, z, t) = ((t * NZ + z) * NY + y) * NX + x;
```

Example B-iii - The SITE calculation

The SITE calculation consists of a series of 6 additions and multiplies of 15 bit numbers that must be performed in sequence. All the operations were performed in a single cycle and the resulting logic had a very long delay. The delay was reduced by splitting the calculation over several cycles. The SITE macro is used nine times on each iteration of the Dirac operator pipeline, so pipelining the macro was the best approach to reducing delay. The macro was split into three stages and pipelined. Each stage performed one addition and one multiply giving well balanced stages.

Further examination of the variables and the values involved revealed that the size of the variables involved in the calculation could be reduced. All of the operands are four bit variables and thus their maximum value is 15. Given that this is the case then the first stage needs to be performed with 8 bit variables, the second with 12 bit variables and the third with 16 bit variables. This reduces the resource requirements of the design.

```
shared expr siteCalc1(x, y, z, t) = (((unsigned 8)(0 @ t) *
(unsigned 8)(0 @ NZ)) + (unsigned 8)(0 @ z));

shared expr siteCalc2() = (((unsigned 12)(0 @ res1) * (unsigned
12)(0 @ NY)) + (unsigned 12)(0 @ yS2));

shared expr siteCalc3() = (((unsigned 16)(0 @ res2) * (unsigned
16)(0 @ NX)) + (unsigned 16)(0 @ xS3));

macro proc siteStage1(x, y, z, t){
 par{
        yS2 = y;
        xS2 = x;
        res1 = siteCalc1(x, y, z, t);
  }
}

macro proc siteStage2(){
 par{
       xS3 = xS2;
       res2 = siteCalc2();
 }
}

macro proc siteStage3(){
 res3 = siteCalc3();
}

macro proc sitePiped(x, y, z, t){
 par{
       siteStage1(x, y, z, t);
       siteStage2();
       siteStage3();
 }
}
```

Example B-iv - Pipelined SITE Calculation

Example B-iv shows the pipelined version of the SITE calculation. To operate, *sitePiped* is called nine times on nine successive cycles and then the pipeline is flushed with two more calls to *sitePiped,* passing any variables as parameters. Only the results on the first nine issues are collected so it is not necessary for the final two calls to have meaningful parameters. The results are collected from *res3* on the appropriate cycles and stored. The pipeline breaks the large delay into three smaller delays whilst only 2 more cycles are required to perform all the SITE calculations for a single point in the lattice.

B.4 Modulus Operator Elimination

The modulus operator is logically complex and incurs a very long logic delay when it is used. It is evaluated using the division operator returning the remainder instead of the division result. FPGA implementations of division are very complex and have very long logic delays. The modulus operator was originally used in the design to calculate the co-ordinates of a lattice point's neighbour. The lattice is wrapped so it is not enough to merely add one or subtract one from a dimension to get a neighbouring point. In the original application the modulus operator was used to handle of this boundary condition.

```
shared expr p1(xIn, NXIn) = (xIn+1) % NXIn;

shared expr m1(xIn, NXIn) = ((xIn+NXIn)-1) % NXIn;
```

Example B-v - Neighbouring Site Calculation Operators

The code in Example B-v shows the original code used to calculate the value of a co-ordinate of plus one or minus one in a single direction within a lattice. Both use a 4 bit modulus operator and were limiting the clock rate of the design to 36 MHz. Eliminating the use of the modulus operator was essential to improve clock rate.

The variables passed to the two expressions conform to a certain rule; that *NXIn* will always be greater than *xIn*. Thus instead of using the modulus operator it was possible to use an if-statement to determine the result. For the *p1* expression the result is *xIn+1* except when *xIn+1* is equal to *NxIn*. In this case the result is zero. For the *m1* expression the result is always *xIn-1* unless *xIn* is zero, in which case the result is *NxIn-1*. In this way if statements were used to eliminate the modulus operator as shown in Example B-vi.

```
macro proc p1(out, xIn, NXIn){
 if((xIn + 1) == NXIn)
       out = 0;
 else
       out = xIn + 1;

macro proc m1(out, xIn, NXIn){
 if(xIn == 0)
       out = NXIn -1;
 else
       out = xIn -1;
```

Example B-vi -P1 and M1 Expressions with Modulus Eliminated

The design could now run at a clock rate of 50 MHz which was a significant improvement over the clock rate of 33 MHz that was possible before the clock rate optimisations were begun.

B.5 Changing to DK3

During the clock rate improvement process a new version of the Handel-C development suite became available. The new version included a retiming optimisation which can dramatically improve the clock rate of a design. Retiming moves logic between cycles in a design in order to balance the delay of adjacent cycles. Retiming does not change the behaviour of the design, but can give significant improvements in clock rate. Retiming improved the clock rate of the log arithmetic Dirac operator design by 30% from 50 MHz to 65 MHz.

B.6 Final Clock Rate Optimisation

At this stage timing analysis indicated that the longest path delay in the design consisted mostly of routing and not logic delay. Routing delay is caused primarily by having long wires between connected parts of the design. It was found that many different distributed RAMs in the design had long routing delays associated with them. These arrays had little to do with each other except that they all shared index variables. Further timing analysis showed that the variable used to index the distributed RAMs had the highest wiring fan-outs in the design.

This indicated that the heavy sharing of the index variables was causing long routing delays. Specifically the variable had to be located centrally on the chip and then connected to many different locations often with long wires. Creating multiple copies of the index variables, each connected to a small number of the RAMs, eliminated the wiring delay.

This optimisation improved the clock rate from a previous 65 MHz to 70 MHz. Timing analysis showed that all the longest paths in the design were now within the log arithmetic units. This indicated that no further clock rate improvements would be possible through optimising the Handel-C code. Discussions with the designer of log arithmetic cores confirmed that 70 MHz is very close to the maximum obtainable clock rate for the cores. At this stage I placed and routed the design for the fastest speed grade FPGA and with a resulting maximum clock rate for the design of 85MHz.

Appendix C

Conjugate Gradient Operator Source Code

This appendix presents the source code for the original C version of the conjugate gradient application, including the source code for the core Dirac operator, which is used in the conjugate gradient application. The source code for the dot-product and vector add-scale operators is also presented along with the main loop that implements the conjugate gradient application. This purpose of this appendix is to illustrate both the conjugate gradient application and the Dirac operator.

The source code is contained in a number of C files; *main.c, latops.c, ops.c gamma.c, ran.c, qcddefs.h* and *qcdtypes.h*. Each file is contained in its own section.

- *Main.c* - Holds the main conjugate gradient application loop
- *Latops.c* – Defines the lattice operations, which are those functions that operate on the entire lattice using the functions in *ops.c* and *gamma.c*
- *Ops.c* – Functions that add, subtract, scale and multiply small complex number matrices.
- *Gamma.c* – All of the *gamma* operators are contained in this file
- *Ran.c* – Functions to generate random numbers for initialising the application

data

- *Qcddefs.h* - The size of the lattice is defined in this file

- *Qcdtypes.h* – The various types used in the application are defined here

The names for the various functions in the source code are different from those used in the thesis. The Dirac operator performed by the LatMulM5Wfv function, the matrix add-scale operator is performed by the LatAddSclWfvWfv function, the matrix scale-add operator is performed by the LatSclWfvAddWfv function and the dot-product operator is performed by the LatDotWfv function.

C.1 *Main.c*

```
#include <stdio.h>
#include <math.h>

#include "qcdtypes.h"
#include "qcddefs.h"
#include "latops.h"

/*
 * main()
 */

main(int argc, char *argv[])
{
  int seqn;
  t_real alpha,beta,res_new,res_old,kappa;

  t_gl3 g[NS][4];
  t_wfv x[NS], y[NS], r[NS], p[NS], tmp1[NS], tmp2[NS];

  /*
   * Initialization.
   */

  InitOffsetArrays();

  RanSetSeed(1);

  kappa = 0.124;

  LatGenGl3(g, 1);

  LatGenZeroWfv(x);
  LatGenZeroWfv(y);
  y[0][0][0].r = 1.0;



/*  START */
  LatCopyWfv(r,y);
  LatCopyWfv(p,y);

  res_old = LatDotWfv(r,r);
  printf("Residual = %e\n",res_old);

  while (res_old > 1.0e-6)
  {

    LatMulM5Wfv(tmp1, kappa, g,   p);
    LatMulM5Wfv(tmp2, kappa, g,tmp1);

    alpha = res_old / LatDotWfv(tmp1, tmp1);

    LatAddSclWfvWfv(r, -alpha, tmp2);
    LatAddSclWfvWfv(x,  alpha,    p);

    res_new = LatDotWfv(r,r);
    printf("Residual = %e\n",res_new);

    beta = res_new / res_old;

    LatSclWfvAddWfv(p, beta, r);

    res_old = res_new;
  }
/*  END   */

  /*
   * All done.
   */

  LatMulM5Wfv(tmp1, kappa, g,   x);
  LatMulM5Wfv(tmp2, kappa, g,tmp1);
  LatAddSclWfvWfv(tmp2, -1.0, y);
  printf("TEST: %e\n",LatDotWfv(tmp2,tmp2));


  exit(0);

}
```

C.2 *Latops.c*

```
/*
 * File: "latops.c"
 *
 *      Routines which evaluate lattice wide operations.
 *
 *
 */

#include <math.h>

#include "qcdtypes.h"
#include "qcddefs.h"
#include "ops.h"
#include "gamma.h"
#include "offsets.h"
#include "latops.h"

/*
 * LatGenGaussWfv()
 */

void LatGenGaussWfv(t_wfv a[NS])
{
  int s;

  for ( s = 0; s < NS; s++)
    GenGaussWfv(a[s]);

  return;
}

/*
 * GenGl3()
 */

void LatGenGl3(t_gl3 g[NS][4], int start_type)
{
  int s, d;

  switch (start_type)
  {

  case 1: /* Cold Start. */
    {
      for (s = 0; s < NS; s++)
      {
        for (d = 0; d < 4; d++)
        {
          GenUnitGl3(g[s][d]);
        }
      }
    }
    break;

  case 2: /* Mixed Start. */
```

```
      {
        for (s = 0; s < NS/2; s++)
        {
          for (d = 0; d < 4; d++)
          {
            GenUnitGl3(g[s][d]);
          }
        }

        for (s = NS/2; s < NS; s++)
        {
          for (d = 0; d < 4; d++)
          {
            GenRandSu3(g[s][d]);
          }
        }
      }
      break;

    case 3: /* Hot Start */
    default:
      {
        for (s = 0; s < NS; s++)
        {
          for (d = 0; d < 4; d++)
          {
            GenRandSu3(g[s][d]);
          }
        }
      }
      break;
    }

    return;
}


/*
 * void LatMulM5Wfv(r, kappa, g, a)
 */

void LatMulM5Wfv(t_wfv r[NS],
                 t_real kappa,
                 t_gl3 g[NS][4],
                 t_wfv a[NS])
{
  int s;

  t_wfv a_spx, a_smx, ga_spx, ga_smx;
  t_wfv a_spy, a_smy, ga_spy, ga_smy;
  t_wfv a_spz, a_smz, ga_spz, ga_smz;
  t_wfv a_spt, a_smt, ga_spt, ga_smt;
  t_wfv tx, ty, tz, tt;
  t_wfv txy, tzt;
  t_wfv tsum, k_tsum;
  t_wfv g5a_s;
  for (s = 0; s < NS; s++)
  {
    G5mG5Gx(a_spx, a[Spx[s]]);
    MulGl3Wfv(ga_spx, g[s][0], a_spx);

    G5pG5Gx(a_smx, a[Smx[s]]);
    MulGl3dWfv(ga_smx, g[Smx[s]][0], a_smx);

    G5mG5Gy(a_spy, a[Spy[s]]);
    MulGl3Wfv(ga_spy, g[s][1], a_spy);

    G5pG5Gy(a_smy, a[Smy[s]]);
    MulGl3dWfv(ga_smy, g[Smy[s]][1], a_smy);

    G5mG5Gz(a_spz, a[Spz[s]]);
    MulGl3Wfv(ga_spz, g[s][2], a_spz);

    G5pG5Gz(a_smz, a[Smz[s]]);
    MulGl3dWfv(ga_smz, g[Smz[s]][2], a_smz);

    G5mG5Gt(a_spt, a[Spt[s]]);
    MulGl3Wfv(ga_spt, g[s][3], a_spt);

    G5pG5Gt(a_smt, a[Smt[s]]);
    MulGl3dWfv(ga_smt, g[Smt[s]][3], a_smt);

    AddWfvWfv(tx, ga_spx, ga_smx);
    AddWfvWfv(ty, ga_spy, ga_smy);
    AddWfvWfv(tz, ga_spz, ga_smz);
    AddWfvWfv(tt, ga_spt, ga_smt);

    AddWfvWfv(txy, tx, ty);
    AddWfvWfv(tzt, tz, tt);

    AddWfvWfv(tsum, txy, tzt);

    MulSclWfv(k_tsum, kappa, tsum);

    G5(g5a_s, a[s]);

    SubWfvWfv(r[s], g5a_s, k_tsum);
  }


  return;
}


/*
 * void LatGenZeroWfv
 */

void LatGenZeroWfv(t_wfv a[NS])
{
  int s;

  for (s = 0; s < NS; s++)
    GenZeroWfv(a[s]);
```

```
}

/*
 * LatCopyWfv
 *      a = b
 */

void LatCopyWfv(t_wfv a[NS], t_wfv b[NS])
{
  int s;

  for ( s = 0; s < NS; s++)
    CopyWfv(a[s], b[s]);

}


/*
 * LatDotWfv
 *      dot product of two Wilson fermion vectors:   a* . b
 */

t_real LatDotWfv(t_wfv a[NS], t_wfv b[NS])
{
  int s;
  t_real dot = 0.0;

  for ( s = 0; s < NS; s++)
    dot += DotWfv(a[s], b[s]);

  return dot;
}

/*
 * LatAddSclWfvWfv
 *        a = a + r * b
 */
void LatAddSclWfvWfv(t_wfv a[NS], t_real r, t_wfv b[NS])
{
  int s;

  for ( s = 0; s < NS; s++)
    AddSclWfvWfv(a[s],r,b[s]);
}

/*
 * LatSclWfvAddWfv
 *        a = r * a + b
 */
void LatSclWfvAddWfv(t_wfv a[NS], t_real r, t_wfv b[NS])
{
  int s;

  for ( s = 0; s < NS; s++)
    SclWfvAddWfv(a[s],r,b[s]);
}
```

C.3 Ops.c

```
/*
 * File: "ops.c"
 *
 *      Routines defining operations on the various primative data
 *      structures used in QCD programs.
 *
 *
 * Change Log:
 *
 *
 */

#include <stdio.h>
#include <stdarg.h>
#include <math.h>

#include "qcdtypes.h"
#include "ops.h"

/*
 * Macro definitions.
 */

#define ONE_ON_SQRT3 ((t_real) 0.57735026918962576451)

/*
 * void RandGauss(x_p, count)
 *
 *      Fills the array pointed to by "x_p" with "count" gaussian
random
 *      variables.
 *      Each variable will have varience 1/2 (i.e. <x^2> = 1/2).
 */

void RandGauss(t_real *x_p,
               int count)
{
  int i;
  t_real theta, r;


  if (count % 2 != 0)
    Err("RandGauss: odd count passed as argument.\n");

  for ( i = 0; i < count; i += 2 )
  {
    theta = 2.0 * M_PI * RandD();
    r = sqrt( - log( RandD() ) );

    *x_p++ = r * cos(theta);
    *x_p++ = r * sin(theta);
  }


  return;
}



/*
 * Gl3 Generation, Print, and Diff Routines.
 */



/*
 * void GenZeroGl3()
 *
 *      Generate a zero 3x3 complex matrix;
 */

void GenZeroGl3(t_gl3 a)
{
  int ci, cj;


  for ( ci = 0; ci < 3; ci++ )
    for ( cj = 0; cj < 3; cj++ )
    {
      a[ci][cj].r = 0.0;
      a[ci][cj].i = 0.0;
    }


  return;
}


/*
 * void GenUnitGl3()
 *
 *      Generate a unit 3x3 complex matrix;
 */

void GenUnitGl3(t_gl3 a)
{
  int ci, cj;


  for ( ci = 0; ci < 3; ci++ )
    for ( cj = 0; cj < 3; cj++ )
    {
      a[ci][cj].r = 0.0;
      a[ci][cj].i = 0.0;
    }

  a[0][0].r = (1.0e0);
  a[1][1].r = (1.0e0);
  a[2][2].r = (1.0e0);


  return;
}
```

```
/*
 * void GenRandGl3()
 *
 *      Generate a random SU(3) matrix;
 */

void GenRandGl3(t_gl3 a)
{
  int ci, cj;


  for ( ci = 0; ci < 3; ci++ )
    for ( cj = 0; cj < 3; cj++ )
    {
      a[ci][cj].r = (t_real) (2.0*RanD() - 1.0);
      a[ci][cj].i = (t_real) (2.0*RanD() - 1.0);
    }


  return;
}

/*
 * Wilson Fermion Vector Operations.
 */



/*
 * void GenZeroWfv()
 *
 *      Generate a zero Wilson Fermion vector.
 */

void GenZeroWfv(t_wfv a)
{
  int d, c;


  for ( d = 0; d < 4; d++ )
    for ( c = 0; c < 3; c++ )
    {
      a[d][c].r = 0.0;
      a[d][c].i = 0.0;
    }

  return;
}



/*
 * void GenGaussWfv()
 *
 *      Generate a Wilson fermion vector with gaussian random
variables.
 */
```

```
void GenGaussWfv(t_wfv a)
{
  RandGauss(&a[0][0].r, (4*3*2));

  return;
}

/*
 * void AddWfvWfv()
 */

void AddWfvWfv(t_wfv r,
               t_wfv a,
               t_wfv b)
{
  int d, c;

  for ( d = 0; d < 4; d++ )
    for ( c = 0; c < 3; c++ )
    {
      r[d][c].r = a[d][c].r + b[d][c].r;
      r[d][c].i = a[d][c].i + b[d][c].i;
    }
  return;
}



/*
 * void SubWfvWfv()
 */

void SubWfvWfv(t_wfv r,
               t_wfv a,
               t_wfv b)
{
  int d, c;

  for ( d = 0; d < 4; d++ )
    for ( c = 0; c < 3; c++ )
    {
      r[d][c].r = a[d][c].r - b[d][c].r;
      r[d][c].i = a[d][c].i - b[d][c].i;
    }
  return;
}



/*
 * void MulGl3Wfv()
 */

void MulGl3Wfv(t_wfv b,
               t_gl3 g,
               t_wfv a)
{
  int c, d;
  for (d = 0; d < 4; d++)
    for (c = 0; c < 3; c++)
    {
      b[d][c].r = g[c][0].r * a[d][0].r
                - g[c][0].i * a[d][0].i
                + g[c][1].r * a[d][1].r
                - g[c][1].i * a[d][1].i
                + g[c][2].r * a[d][2].r
                - g[c][2].i * a[d][2].i;
      b[d][c].i = g[c][0].r * a[d][0].i
                + g[c][0].i * a[d][0].r
                + g[c][1].r * a[d][1].i
                + g[c][1].i * a[d][1].r
                + g[c][2].r * a[d][2].i
                + g[c][2].i * a[d][2].r;
    }

  return;
}



/*
 * void MulGl3dWfv()
 */

void MulGl3dWfv(t_wfv b,
                t_gl3 g,
                t_wfv a)
{
  int c, d;


  for (d = 0; d < 4; d++)
    for (c = 0; c < 3; c++)
    {
      b[d][c].r = g[0][c].r * a[d][0].r
                + g[0][c].i * a[d][0].i
                + g[1][c].r * a[d][1].r
                + g[1][c].i * a[d][1].i
                + g[2][c].r * a[d][2].r
                + g[2][c].i * a[d][2].i;
      b[d][c].i = g[0][c].r * a[d][0].i
                - g[0][c].i * a[d][0].r
                + g[1][c].r * a[d][1].i
                - g[1][c].i * a[d][1].r
                + g[2][c].r * a[d][2].i
                - g[2][c].i * a[d][2].r;
    }


  return;
}



/*
 * void MulSclWfv()
 */

void MulSclWfv(t_wfv b,
               t_real k,
               t_wfv a)
{
  int c, d;


  for ( d = 0; d < 4; d++ )
    for ( c = 0; c < 3; c++ )
    {
      b[d][c].r = k * a[d][c].r;
      b[d][c].i = k * a[d][c].i;
    }


  return;
}

/*     ============================================================
*/


/*
 * Copy
 *     a = b
 */

void CopyWfv(t_wfv a, t_wfv b)
{
  int d, c;

  for ( d = 0; d < 4; d++ )
    for ( c = 0; c < 3; c++ )
    {
      a[d][c].r = b[d][c].r;
      a[d][c].i = b[d][c].i;
    }
}


/*
 * DotWfv
 */

t_real DotWfv(t_wfv a, t_wfv b)
{
  t_real dot = 0.0;
  int d, c;

  for ( d = 0; d < 4; d++ )
    for ( c = 0; c < 3; c++ )
      dot += a[d][c].r * b[d][c].r + a[d][c].i * b[d][c].i;

  return dot;
}

/*
```

```
 * AddSclWfvWfv
 *        a = a + r * b
 */
void AddSclWfvWfv(t_wfv a, t_real r, t_wfv b)
{
  int d, c;

  for ( d = 0; d < 4; d++ )
    for ( c = 0; c < 3; c++ )
    {
      a[d][c].r += r * b[d][c].r;
      a[d][c].i += r * b[d][c].i;
    }
}


/*
 * SclWfvAddWfv
 *        a = r * a +  b
 */
void SclWfvAddWfv(t_wfv a, t_real r, t_wfv b)
{
  int d, c;

  for ( d = 0; d < 4; d++ )
    for ( c = 0; c < 3; c++ )
    {
      a[d][c].r = r * a[d][c].r + b[d][c].r;
      a[d][c].i = r * a[d][c].i + b[d][c].i;
    }
}
```

C.4 Gamma.c

```
/*
 * File: "gamma.c"
 *
 *      Gamma matrix operations.
 *
 *
 * Change Log:
 *
 *
 */

#include "qcdtypes.h"
#include "gamma.h"



/*
 * Gamma matrix conventions:
 *
 *      Gx      = \rho_1 \sigma_1       = |           1 |
 *                                        |        1    |
 *                                        |     1       |
 *                                        |  1          |
 *
 *      Gy      = \rho_1 \sigma_2       = |          -i |
 *                                        |        i    |
 *                                        |    -i       |
 *                                        |  i          |
 *
 *      Gz      = \rho_1 \sigma_3       = |        1    |
 *                                        |          -1 |
 *                                        |  1          |
 *                                        |    -1       |
 *
 *      Gt      = \rho_2                = |       -i    |
 *                                        |          -i |
 *                                        |  i          |
 *                                        |     i       |
 *
 *      G5      = \rho_3                = |  1          |
 *                                        |     1       |
 *                                        |       -1    |
 *                                        |          -1 |
 */



/*
 * void Gx()
 */

void Gx(t_wfv r,
        t_wfv a)
{
  int c;
  for ( c = 0; c < 3; c++ )
  {
    r[0][c].r = + a[3][c].r;
    r[0][c].i = + a[3][c].i;
    r[1][c].r = + a[2][c].r;
    r[1][c].i = + a[2][c].i;
    r[2][c].r = + a[1][c].r;
    r[2][c].i = + a[1][c].i;
    r[3][c].r = + a[0][c].r;
    r[3][c].i = + a[0][c].i;
  }

  return;
}



/*
 * void Gy()
 */

void Gy(t_wfv r,
        t_wfv a)
{
  int c;

  for ( c = 0; c < 3; c++ )
  {
    r[0][c].r = + a[3][c].i;
    r[0][c].i = - a[3][c].r;
    r[1][c].r = - a[2][c].i;
    r[1][c].i = + a[2][c].r;
    r[2][c].r = + a[1][c].i;
    r[2][c].i = - a[1][c].r;
    r[3][c].r = - a[0][c].i;
    r[3][c].i = + a[0][c].r;
  }

  return;
}



/*
 * void Gz()
 */

void Gz(t_wfv r,
        t_wfv a)
{
  int c;

  for ( c = 0; c < 3; c++ )
  {
    r[0][c].r = + a[2][c].r;
    r[0][c].i = + a[2][c].i;
    r[1][c].r = - a[3][c].r;
    r[1][c].i = - a[3][c].i;
    r[2][c].r = + a[0][c].r;
    r[2][c].i = + a[0][c].i;
    r[3][c].r = - a[1][c].r;
    r[3][c].i = - a[1][c].i;
  }

  return;
}



/*
 * void Gt()
 */

void Gt(t_wfv r,
        t_wfv a)
{
  int c;

  for ( c = 0; c < 3; c++ )
  {
    r[0][c].r = + a[2][c].i;
    r[0][c].i = - a[2][c].r;
    r[1][c].r = + a[3][c].i;
    r[1][c].i = - a[3][c].r;
    r[2][c].r = - a[0][c].i;
    r[2][c].i = + a[0][c].r;
    r[3][c].r = - a[1][c].i;
    r[3][c].i = + a[1][c].r;
  }

  return;
}



/*
 * void G5()
 */

void G5(t_wfv r,
        t_wfv a)
{
  int d, c;


  for ( c = 0; c < 3; c++ )
  {
    r[0][c].r = + a[0][c].r;
    r[0][c].i = + a[0][c].i;
    r[1][c].r = + a[1][c].r;
    r[1][c].i = + a[1][c].i;
    r[2][c].r = - a[2][c].r;
    r[2][c].i = - a[2][c].i;
    r[3][c].r = - a[3][c].r;
    r[3][c].i = - a[3][c].i;
  }
  return;
}



/*
 * void IpGx()
 */

void IpGx(t_wfv r,
             t_wfv a)
{
  int c;

  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r + a[3][c].r;
    r[0][c].i = a[0][c].i + a[3][c].i;
    r[1][c].r = a[1][c].r + a[2][c].r;
    r[1][c].i = a[1][c].i + a[2][c].i;

    r[2][c].r = +r[1][c].r;
    r[2][c].i = +r[1][c].i;
    r[3][c].r = +r[0][c].r;
    r[3][c].i = +r[0][c].i;
  }

  return;
}



/*
 * void ImGx()
 */

void ImGx(t_wfv r,
             t_wfv a)
{
  int c;

  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r - a[3][c].r;
    r[0][c].i = a[0][c].i - a[3][c].i;
    r[1][c].r = a[1][c].r - a[2][c].r;
    r[1][c].i = a[1][c].i - a[2][c].i;

    r[2][c].r = -r[1][c].r;
    r[2][c].i = -r[1][c].i;
    r[3][c].r = -r[0][c].r;
    r[3][c].i = -r[0][c].i;
  }

  return;
}
```

```
/*
 * void IpGy()
 */

void IpGy(t_wfv r,
             t_wfv a)
{
  int c;

  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r + a[3][c].i;
    r[0][c].i = a[0][c].i - a[3][c].r;
    r[1][c].r = a[1][c].r - a[2][c].i;
    r[1][c].i = a[1][c].i + a[2][c].r;

    r[2][c].r = +r[1][c].i;
    r[2][c].i = -r[1][c].r;
    r[3][c].r = -r[0][c].i;
    r[3][c].i = +r[0][c].r;
  }

  return;
}



/*
 * void ImGy()
 */

void ImGy(t_wfv r,
             t_wfv a)
{
  int c;

  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r - a[3][c].i;
    r[0][c].i = a[0][c].i + a[3][c].r;
    r[1][c].r = a[1][c].r + a[2][c].i;
    r[1][c].i = a[1][c].i - a[2][c].r;

    r[2][c].r = -r[1][c].i;
    r[2][c].i = +r[1][c].r;
    r[3][c].r = +r[0][c].i;
    r[3][c].i = -r[0][c].r;
  }

  return;
}



/*
 * void IpGz()
 */
```

```
void IpGz(t_wfv r,
             t_wfv a)
{
  int c;

  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r + a[2][c].r;
    r[0][c].i = a[0][c].i + a[2][c].i;
    r[1][c].r = a[1][c].r - a[3][c].r;
    r[1][c].i = a[1][c].i - a[3][c].i;

    r[2][c].r = +r[0][c].r;
    r[2][c].i = +r[0][c].i;
    r[3][c].r = -r[1][c].r;
    r[3][c].i = -r[1][c].i;
  }

  return;
}



/*
 * void ImGz()
 */

void ImGz(t_wfv r,
             t_wfv a)
{
  int c;

  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r - a[2][c].r;
    r[0][c].i = a[0][c].i - a[2][c].i;
    r[1][c].r = a[1][c].r + a[3][c].r;
    r[1][c].i = a[1][c].i + a[3][c].i;

    r[2][c].r = -r[0][c].r;
    r[2][c].i = -r[0][c].i;
    r[3][c].r = +r[1][c].r;
    r[3][c].i = +r[1][c].i;
  }

  return;
}



/*
 * void IpGt()
 */

void IpGt(t_wfv r,
             t_wfv a)
{
  int c;
  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r + a[2][c].i;
    r[0][c].i = a[0][c].i - a[2][c].r;
    r[1][c].r = a[1][c].r + a[3][c].i;
    r[1][c].i = a[1][c].i - a[3][c].r;

    r[2][c].r = -r[0][c].i;
    r[2][c].i = +r[0][c].r;
    r[3][c].r = -r[1][c].i;
    r[3][c].i = +r[1][c].r;
  }

  return;
}



/*
 * void ImGt()
 */

void ImGt(t_wfv r,
             t_wfv a)
{
  int c;

  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r - a[2][c].i;
    r[0][c].i = a[0][c].i + a[2][c].r;
    r[1][c].r = a[1][c].r - a[3][c].i;
    r[1][c].i = a[1][c].i + a[3][c].r;

    r[2][c].r = +r[0][c].i;
    r[2][c].i = -r[0][c].r;
    r[3][c].r = +r[1][c].i;
    r[3][c].i = -r[1][c].r;
  }

  return;
}



/*
 * void G5pG5Gx()
 */

void G5pG5Gx(t_wfv r,
             t_wfv a)
{
  int c;

  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r + a[3][c].r;
    r[0][c].i = a[0][c].i + a[3][c].i;
    r[1][c].r = a[1][c].r + a[2][c].r;
    r[1][c].i = a[1][c].i + a[2][c].i;

    r[2][c].r = -r[1][c].r;
    r[2][c].i = -r[1][c].i;
    r[3][c].r = -r[0][c].r;
    r[3][c].i = -r[0][c].i;
  }

  return;
}



/*
 * void G5mG5Gx()
 */

void G5mG5Gx(t_wfv r,
             t_wfv a)
{
  int c;

  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r - a[3][c].r;
    r[0][c].i = a[0][c].i - a[3][c].i;
    r[1][c].r = a[1][c].r - a[2][c].r;
    r[1][c].i = a[1][c].i - a[2][c].i;

    r[2][c].r = +r[1][c].r;
    r[2][c].i = +r[1][c].i;
    r[3][c].r = +r[0][c].r;
    r[3][c].i = +r[0][c].i;
  }

  return;
}



/*
 * void G5pG5Gy()
 */

void G5pG5Gy(t_wfv r,
             t_wfv a)
{
  int c;

  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r + a[3][c].i;
    r[0][c].i = a[0][c].i - a[3][c].r;
    r[1][c].r = a[1][c].r - a[2][c].i;
    r[1][c].i = a[1][c].i + a[2][c].r;

    r[2][c].r = -r[1][c].i;
    r[2][c].i = +r[1][c].r;
    r[3][c].r = +r[0][c].i;
    r[3][c].i = -r[0][c].r;
  }

  return;
}



/*
 * void G5mG5Gy()
 */

void G5mG5Gy(t_wfv r,
             t_wfv a)
{
  int c;

  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r - a[3][c].i;
    r[0][c].i = a[0][c].i + a[3][c].r;
    r[1][c].r = a[1][c].r + a[2][c].i;
    r[1][c].i = a[1][c].i - a[2][c].r;

    r[2][c].r = +r[1][c].i;
    r[2][c].i = -r[1][c].r;
    r[3][c].r = -r[0][c].i;
    r[3][c].i = +r[0][c].r;
  }

  return;
}



/*
 * void G5pG5Gz()
 */

void G5pG5Gz(t_wfv r,
             t_wfv a)
{
  int c;

  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r + a[2][c].r;
    r[0][c].i = a[0][c].i + a[2][c].i;
    r[1][c].r = a[1][c].r - a[3][c].r;
    r[1][c].i = a[1][c].i - a[3][c].i;

    r[2][c].r = -r[0][c].r;
    r[2][c].i = -r[0][c].i;
    r[3][c].r = +r[1][c].r;
    r[3][c].i = +r[1][c].i;
  }

  return;
}
```

```
/*
 * void G5mG5Gz()
 */

void G5mG5Gz(t_wfv r,
             t_wfv a)
{
  int c;

  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r - a[2][c].r;
    r[0][c].i = a[0][c].i - a[2][c].i;
    r[1][c].r = a[1][c].r + a[3][c].r;
    r[1][c].i = a[1][c].i + a[3][c].i;

    r[2][c].r = +r[0][c].r;
    r[2][c].i = +r[0][c].i;
    r[3][c].r = -r[1][c].r;
    r[3][c].i = -r[1][c].i;
  }

  return;
}



/*
 * void G5pG5Gt()
 */

void G5pG5Gt(t_wfv r,
             t_wfv a)
{
  int c;

  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r + a[2][c].i;
    r[0][c].i = a[0][c].i - a[2][c].r;
    r[1][c].r = a[1][c].r + a[3][c].i;
    r[1][c].i = a[1][c].i - a[3][c].r;

    r[2][c].r = +r[0][c].i;
    r[2][c].i = -r[0][c].r;
    r[3][c].r = +r[1][c].i;
    r[3][c].i = -r[1][c].r;
  }

  return;
}



/*
 * void G5mG5Gt()
 */

void G5mG5Gt(t_wfv r,
             t_wfv a)
{
  int c;

  for (c = 0; c < 3; c++)
  {
    r[0][c].r = a[0][c].r - a[2][c].i;
    r[0][c].i = a[0][c].i + a[2][c].r;
    r[1][c].r = a[1][c].r - a[3][c].i;
    r[1][c].i = a[1][c].i + a[3][c].r;

    r[2][c].r = -r[0][c].i;
    r[2][c].i = +r[0][c].r;
    r[3][c].r = -r[1][c].i;
    r[3][c].i = +r[1][c].r;
  }

  return;
}
```

C.5 Ran.c

```
/*
 * File: "sys/libutl/ran.c"
 *
 *      Random number routines.
 *
 *
 * Change Log:
 *
 *
 */

#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#include "misc.h"
#include "err.h"
#include "ran.h"



/*
 * Specify Random Number Generator to use
 *
 *  Define Ran_USE_DRAND48 to use system supplied drand48() routines
 *  Define Ran_USE_RAN32 to use portable Ran32 generator
 */

#define Ran_USE_DRAND48



/*
 * void RanSetSeed(int use_clock)
 *
 *      Sets the current value of the random number seed.
 *
 */

void RanSetSeed(int use_clock)
{
  time_t time_in_secs;
  unsigned int secs, num1, num2, pattern1, pattern2;
  unsigned int i;
  t_seed seed;



  /*
   * Check that an int contains 4 bytes, and that type t_time
contains 4 bytes
   */

  if ( sizeof(int) < 4 || sizeof(time_t) < 4 )
  {
    Err("RanSetSeed: int, time_t byte size is too small.\n");
  }


  /*
   * First step is to generate a 32 bit pattern in variable
"pattern1",
   * either from the clock or from a predefined pattern.
   *
   * We then generate a second pattern, "pattern2" using a mod 32
   * linear congruential generator from "pattern1" for when we need
   * more than 32 bits of seed.
   */

  if ( use_clock )
  {
    /*
     * Initialization from clock.
     *
     *  When seed is taken from the clock, the original 32-bit
     *  sequence from the clock in num1
     *
     *          31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
     *          15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
     *
     *  is reversed in order to produce num2:
     *
     *           0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
     *          16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
     *
     *  num1 and num2 are then combined to produce num3:
     *
     *          31  1 29  3  4 26  6 24 23  9 21 11 12 18 14 16
     *          15 17 13 19 20 10 22  8  7 25  5 27 28  2 30  0
     */

    time_in_secs = time((void *) 0);
    secs = ((unsigned int) time_in_secs) & 0xffffffff;

    num1 = secs;
    num2 = secs & 0x00000001;

    for (i = 1; i < 32; i++)
    {
      num2 <<= 1;
      secs >>= 1;
      num2 |= secs & 0x00000001;
    }

    pattern1 = (num1 & (0xa5a5a5a5)) | (num2 & ~(0xa5a5a5a5));
  }
  else
  {
    pattern1 = 0x9876abcd;
  }

  pattern2 = (pattern1*1021021 + 345897122) & 0xffffffff;
  /*
   * Now, use pattern1 and pattern2 to store the seed.
   */

#ifdef Ran_USE_DRAND48

  seed.data_s[0] = pattern1 & 0xffff;
  seed.data_s[1] = pattern2 & 0xffff;
  seed.data_s[2] = (pattern1 >> 16) & 0xffff;
  seed.data_s[3] = 0;

#else
#ifdef Ran_USE_RAN32

  seed.data_i[0] = pattern1;
  seed.data_i[1] = 0;

#else

  Err("No seed generator specified.\n");

#endif
#endif


  /*
   * Now set the value of the seed.
   */

  RanResetSeed(&seed);

  return;
}



/*
 * void RanResetSeed(t_seed *seed_p)
 *
 *      Resets the random number seed to the value passed in in
"seed_p".
 */

void RanResetSeed(t_seed *seed_p)
{
  unsigned short *data_p;


  /*
   * Do the reset.
   */

#ifdef Ran_USE_DRAND48

  seed48( seed_p->data_s );

#else
#ifdef Ran_USE_RAN32
```

```
  Ran32_SetSeed(seed_p->data_i[0]);

#else

  Err("No seed generator specified.\n");

#endif
#endif

}



/*
 * void RanGetSeed(t_seed *seed_p)
 *
 *       Fetch the current random number seed and return it in the
structure
 *       pointed to by "seed_p".  This routine makes no change to the
current
 *       seed.
 */

void RanGetSeed(t_seed *seed_p)
{
  unsigned short *data_p;
  int i;



  /*
   * Get the current value of the seed.
   */

#ifdef Ran_USE_DRAND48

  /*
   * NOTE, for drand48, getting the seed changes it, so must reset
after
   */

  data_p = seed48(seed_p->data_s);

  for ( i = 0; i < 3; i ++ )
    seed_p->data_s[i] = data_p[i];
  seed_p->data_s[3] = 0;

  data_p = seed48(seed_p->data_s);

#else
#ifdef Ran_USE_RAN32

  seed_p->data_i[0] = Ran32_GetSeed();
  seed_p->data_i[1] = 0;

#else

  Err("No seed generator specified.\n");
```

```
#endif
#endif


  return;
}



/*
 * void RanCopySeed()
 *
 *      Copies a random number seed.
 */

void RanCopySeed(t_seed *dst_seed_p, t_seed *src_seed_p)
{
  int i;


#ifdef Ran_USE_DRAND48

  dst_seed_p->data_s[0] = src_seed_p->data_s[0];
  dst_seed_p->data_s[1] = src_seed_p->data_s[1];
  dst_seed_p->data_s[2] = src_seed_p->data_s[2];
  dst_seed_p->data_s[3] = 0;

#else
#ifdef Ran_USE_RAN32

  dst_seed_p->data_i[0] = src_seed_p->data_i[0];
  dst_seed_p->data_i[1] = 0;

#else

  Err("No seed generator specified.\n");

#endif
#endif


  return;
}



/*
 * char *RanSeedText(seed)
 *
 *      Formats its argument as a random number seed and returns a
pointer
 *      to the formatted string.  Designed to be used as an argument
in a
 *      printf statement.
 */

char *RanSeedText(t_seed *seed_p)
{
  static char RanSeedString[64];


#ifdef Ran_USE_DRAND48

  sprintf(RanSeedString, "%4.4x %4.4x %4.4x",
          seed_p->data_s[0], seed_p->data_s[1], seed_p->data_s[2]);

#else
#ifdef Ran_USE_RAN32

  sprintf(RanSeedString, "%8.8x",
          seed_p->data_i[0]);

#else

  Err("No seed generator specified.\n");

#endif
#endif

  return RanSeedString;
}



/*
 * unsigned int RanU()
 *
 *      Returns an unsigned int uniformly distributed in range [0,
2^31-1]
 */

unsigned int RanU()
{

#ifdef Ran_USE_DRAND48

  return ((unsigned int) lrand48());

#else
#ifdef Ran_USE_RAN32

  return Ran32_UInt();

#else

  Err("No seed generator specified.\n");

#endif
#endif

}



/*
 * int RanModN()
 *
 *      Returns an int uniformly in range [0, n-1]
 */

int RanModN(int n)
{

  return RanU() % n;

}



/*
 * int RanF()
 *
 *      Returns a float uniformly distributed in range [0.0, 1.0]
 */

float RanF()
{

#ifdef Ran_USE_DRAND48

  return (float) drand48();

#else
#ifdef Ran_USE_RAN32

  return (float) Ran32_Double();

#else

  Err("No seed generator specified.\n");

#endif
#endif

}



/*
 * int RanD()
 *
 *      Returns a double uniformly distributed in range [0.0, 1.0]
 */

double RanD()
{

#ifdef Ran_USE_DRAND48

  return drand48();

#else
#ifdef Ran_USE_RAN32

  return Ran32_Double();
```

```
#else

  Err("No seed generator specified.\n");

#endif
#endif

}



/*
 * 32 Bit Random Number Generator
 *
 *  Linear Congruential Generator
 *   x -> (a x + c) % m
 *  where
 *   m = 2^32
 *   a = (4*3*5*7*11*13*17*19+1)
 *   c = (23*29*37)
 *  Notes: (a-1) must be a multiple of 4
 *   c must be relatively prime to m
 */

#define Ran32_M_A              19399381
#define Ran32_M_C              24679
#define Ran32_M_Scale                ((double) 1.0)/((double)
4294967295.0)
#define Ran32_M_Start_Seed           0x1324acbd

static unsigned int Ran32_Seed = Ran32_M_Start_Seed;



/*
 * unsigned int Ran32_UInt()
 *
 *  returns a random integer in the range [0, 2^31-1] inclusive
 */

unsigned int Ran32_UInt()
{
  Ran32_Seed = (Ran32_M_A * Ran32_Seed + Ran32_M_C) & 0xffffffff;

  return Ran32_Seed % 0x7fffffff;
}



/*
 * double Ran32_Double()
 *
 *  returns a double in the range [0.0, 1.0] inclusive
 */

double Ran32_Double()
{
  Ran32_Seed = (Ran32_M_A * Ran32_Seed + Ran32_M_C) & 0xffffffff;
  return ((double) Ran32_Seed) * Ran32_M_Scale;
}



/*
 * void Ran32_SetSeed(unsigned int seed)
 *
 *  Sets the random number seed.
 */

void Ran32_SetSeed(unsigned int seed)
{
  Ran32_Seed = (seed & 0xffffffff);
}



/*
 * unsigned int Ran32_GetSeed()
 *
 *  Returns the current Ran32 seed.
 */

unsigned int Ran32_GetSeed()
{
  return Ran32_Seed;
}



/*
 * void Ran32_TestFns()
 *
 *  A routine which tests Ran32 random number seed functions.
 */

#define Ran32_M_MomentCnt     20
#define Ran32_M_MomentIters   100000

void Ran32_TestFns()
{
  unsigned int i, m;
  double r, t, x[Ran32_M_MomentCnt];


#ifdef Ran_USE_DRAND48
  return;
#endif


  /*
   * Banner
   */

  printf("=== Test of Ran32 Functions\n\n");


  /*
```

```
     * unsigned int type must be exactly 32 bits.
     */

    if ( sizeof(unsigned int) < 4 )
    {
      Err("  Unsigned int must be at least 4 bytes, 32 bits.\n");
    }


    /*
     * Test Ran32_M_Scale is correct.
     */

    i = 0xffffffff;
    r = ((double) i) * ((double) Ran32_M_Scale);

    printf("  largest int random number should be: 4294967295\n");
    printf("  largest int random number is:        %u\n", i);
    printf("\n");

    printf("  (largest int random number * Ran32_M_Scale - 1.0) should
  be: %le\n", 0.0);
    printf("  (largest int random number * Ran32_M_Scale - 1.0) is:
  %le\n", r-1.0);
    printf("\n");

    if ( r != ((double) 1.0) )
    {
      Err("  Ran32_M_Scale factor incorrect.\n");
    }


    /*
     * Generate Some Random Integers
     */

    printf("  Random Integers\n");
    for ( i = 0; i < 10; i++ )
      printf("    %8.8x\n", Ran32_UInt());
    printf("\n");


    /*
     * Generate Some Random Doubles
     */

    printf("  Random Doubles\n");
    for ( i = 0; i < 10; i++ )
      printf("    %lf\n", Ran32_Double());
    printf("\n");


    /*
     * Calculate Moments
     */

    for ( m = 0; m < Ran32_M_MomentCnt; m++ )
      x[m] = 0.0;
    for ( i = 0; i < Ran32_M_MomentIters; i++ )
    {
      t = r = Ran32_Double();
      for ( m = 0; m < 10; m++ )
      {
        x[m] += t;
        t *= r;
      }
    }

    for ( m = 0; m < 10; m++ )
      x[m] /= Ran32_M_MomentIters;

    printf("  Moment test: \n");

    for ( m = 0; m < 10; m++ )
    {
      printf("    %2d <x^%2d>:  %lf,  should be 1.0\n", m+2, m+1,
  (m+2)*x[m]);
    }
    printf("\n");


    /*
     * If we get to here then everything is ok.
     */

    printf("  Tests successful.\n\n");
    fflush(stdout);

    return;
  }
```

C.6 Qcddefs.h

```
/*
 * File: "qcddefs.h"
 *
 *      Macro definitions defining lattice sizes and other compile
time
 *      parameters defining simulations to be run.
 *
 *
 * Change Log:
 *
 *
 */

#define NX 4
#define NY 4
#define NZ 4
#define NT 4

#define NS (NX*NY*NZ*NT)


/*
 * End of "qcddefs.h"
 */
```

C.7 *Qcdtypes.h*

```
/*
 * File: "qcdtypes.h"
 *
 *      Basic QCD data type definitions.
 *
 *
 * Change Log:
 *
 *
 */



/*
 * Primitive types used in QCD programs:
 *
 *  t_char          Character data.
 *  t_int           Integer Numbers.
 *      t_real          Real Numbers.
 *
 *      t_complex       Complex Numbers.
 *
 *      t_su2           SU(2) matrix.
 *      t_gl3           3x3 complex matrix, or SU(3) matrix.
 *      t_wfv           Wilson Fermion vector.
 *  t_mom           Momentum vector for Hybrid MC.
 *
 * The following typedefs are needed to generate fast code.
 *
 *  t_fcomplex      Complex Numbers.
 *  t_fgl3          3x3 complex matrix, or SU(3) matrix.
 *  t_fwfv          Wilson Fermion vector.
 */


typedef char            t_char;
typedef int         t_int;
typedef double          t_real;


typedef struct s_complex
{
  double r;
  double i;
}
t_complex;

typedef t_real          t_su2 [4];
typedef t_real          t_mom [8];
typedef t_complex       t_gl3 [3][3];
typedef t_complex       t_wfv [4][3];

typedef t_real          t_fcomplex [2];
typedef t_real          t_fgl3 [18];
typedef t_real          t_fwfv [24];
```

```
/*
 * Macros to cast between different forms of equal types.
 */

#define GL3(a)          ((t_complex (*)[3]) a)
#define FGL3(a)         ((t_real *) a)
#define WFV(a)          ((t_complex (*)[3]) a)
#define FWFV(a)         ((t_real *) a)

/*
 * End of "qcdtypes.h"
 */
```

Appendix D

Handel-C code for double precision conjugate gradient implementation

This appendix contains the Handel-C source code for the double precision Dirac operator designs. The source code is included to illustrate the double precision Dirac operator design, and to illustrate how Handel-C can be used to create FPGA based hardware using external floating point arithmetic cores.

The source code is split in several files.

- *Main.hcc* – The main loop that runs the conjugate gradient application.
- *Gamma.hcc* – The combined *gamma-mul* operator is defined in this file along with operators to efficiently add, subtract and scale small complex number matrices. Three operators the run three of the four stages in the Dirac operator pipeline, using operators defined in *gamma.hcc* are also defined here.
- *CG_ops.hcc* – The *dot-product* and *vector add-scale* operators used by the main conjugate gradient application loop are defined here.

- *SRAM_functions.hcc* - All operators related to moving data to and from memory are defined here, including an operator that retrieves all data required for a single iteration of the Dirac operator pipeline.

- *Types.hch* - The various types of on-chip RAMs used in the design are defined in this file.

- *Variables.hch* – For easy access all the global variables used in the conjugate gradient application are defined in this file. This file is included once in the *main.hcc* file.

D.1 Main.hcc

```
#include "defs.h"
#include "types.hch"

#ifndef SIMULATE
#define TARGET_EDIF

#include "plxpci.h"

extern interface BUFG (unsigned 1 O) mclkBUFG1X (unsigned  1 I);
set clock = internal mclkBUFG1X.O with{rate = 80};

#else

set clock = external("D");

#endif

//#define PERF_COUNTERS

#include "sram0.h"
#include "sram1.h"
#include "sram2.h"
#include "sram3.h"
#include "sram4.h"
#include "sram5.h"

#include "sramaccess.c"

#include "ops.hcc"

//Global application variables

unsigned 1 runLoops;

macro expr SITE(x, y, z, t) =
     ((((unsigned NSWIDTH)(0 @t) *

     (unsigned NSWIDTH)(0 @ NZ) + (unsigned NSWIDTH)(0 @ z))

     * (unsigned NSWIDTH)(0 @ NY) + (unsigned NSWIDTH)(0 @ y))

     * (unsigned NSWIDTH)(0 @ NX) + (unsigned NSWIDTH)(0 @ x));

macro expr LOWERDOUBLE(a) = (a<-32);
macro expr UPPERDOUBLE(a) = (a\\32);
macro expr LOWERINT(a) = (a<-32);
macro expr UPPERINT(a) = (a\\32);
macro expr BUILDINT64(x, y) = (unsigned 64)(x @ y);

macro expr G_ADDRESS(s, t, a) =
    (((s * 36) + (t * 9) + (unsigned 20)(0 @ a)) * 2);

macro expr Y_ADDRESS(wfv, element) =
    (((wfv * 12) + (unsigned 20)(0 @ element)) * 2);

macro expr G_ADDRESS_P1(s, t, a) =
    (G_ADDRESS(s, t, (unsigned 20)(0 @ a)) + 1);

macro expr Y_ADDRESS_P1(wfv, element) =
    ((Y_ADDRESS(wfv, element)) + 1);

#include "variables.hch"
#include "gamma.hcc"
#include "sramfunctions.hcc"
#include "cg ops.hcc"

macro proc CalculateOffsets()
{
 s = SITE(x, y, z, t);

 Spx = SITE(xp1, y, z, t);
 Spy = SITE(x, yp1, z, t);
 Spz = SITE(x, y, zp1, t);
 Spt = SITE(x, y, z, tp1);

 Smx = SITE(xm1, y, z, t);
 Smy = SITE(x, ym1, z, t);
 Smz = SITE(x, y, zm1, t);
 Smt = SITE(x, y, z, tm1);
}

shared expr GetParmsMult(s, t) = (s * (unsigned NSWIDTH)(0 @ t));

shared expr IncOperands(op) = op + 1;

macro proc GetParameters()
{
 unsigned 32 h, l;
 unsigned 20 operands;

 operands = PARMS_BASE;

 l = ReadBank2((unsigned 20)(0 @ (operands)));
 operands = IncOperands(operands);
 h = ReadBank2((unsigned 20)(0 @ operands));
 kappa = BUILDINT64(h, l);

 NS = 1; //Init to one

 operands = IncOperands(operands);
 h = ReadBank2((unsigned 20)(0 @ operands));
 NX = h<-XWIDTH;
 NS = GetParmsMult(NS, NX);

 operands = IncOperands(operands);
 h = ReadBank2((unsigned 20)(0 @ operands));
 NY = h<-YWIDTH;
 NS = GetParmsMult(NS, NY);

 operands = IncOperands(operands);
 h = ReadBank2((unsigned 20)(0 @ operands));
 NZ = h<-ZWIDTH;
 NS = GetParmsMult(NS, NZ);

 operands = IncOperands(operands);
 h = ReadBank2((unsigned 20)(0 @ operands));
 NT = h<-TWIDTH;
 NS = GetParmsMult(NS, NT);

 operands = IncOperands(operands);
 numIter = ReadBank2((unsigned 20)(0 @ operands));

 operands = IncOperands(operands);
 l = ReadBank2((unsigned 20)(0 @ (operands)));
 operands = IncOperands(operands);
 h = ReadBank2((unsigned 20)(0 @ operands));
 threshold = BUILDINT64(h, l);
}

macro proc WriteParameters()
{
 unsigned 20 operands;

 operands = PARMS_BASE;

 WriteBank5((unsigned 20)(0 @ (operands)), kappa<-32);
 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ operands), kappa\\32);

 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ operands), (unsigned 32)(0 @ NX));

 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ operands), (unsigned 32)(0 @ NY));

 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ operands), (unsigned 32)(0 @ NZ));

 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ operands), (unsigned 32)(0 @ NT));

 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ operands), numIter);

 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ (operands)), threshold<-32);
 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ operands), threshold\\32);

 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ (operands)), res_old<-32);
 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ operands), res_old\\32);

 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ (operands)), alpha<-32);
 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ operands), alpha\\32);

 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ (operands)), beta<-32);
 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ operands), beta\\32);
 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ (operands)),
             res_new<-32);
 operands = IncOperands(operands);
 WriteBank5((unsigned 20)(0 @ operands), res_new\\32);
}

void LatMulM5Wfv_Piped(unsigned 20 rOffset, unsigned 20 oOffset){
 unsigned 1 runStage1Loop, runStage2Loop, runStage3Loop;
 unsigned 1 delayStage1Loop, delayStage2Loop, delayStage3Loop;
 unsigned 1 doneReadOp, doneStage1, doneStage2, doneStage3;
 signal unsigned 1 iterationComplete;
 unsigned 1 stage1Active, stage2Active, stage3Active;
 unsigned 20 resOffset, opOffset;

 //piped running
 par{
  runReadOpLoop = 1;
  runStage1Loop = 1;
  runStage2Loop = 1;
  runStage3Loop = 1;
  delayStage1Loop = 1;
  delayStage2Loop = 1;
  delayStage3Loop = 1;

  doneReadOp = 1;
  doneStage1 = 1;
  doneStage2 = 1;
  doneStage3 = 1;

  opOffset = oOffset;
  resOffset = rOffset;

  x = 0; y = 0; z = 0; t = 0; iter = 0;
  stage1Active = 0; stage2Active = 0; stage3Active = 0;
 }
  par{
   seq{
    do{
     iterationComplete = !(doneReadOp & doneStage1 & doneStage2 &
doneStage3);
    }while(runStage3Loop);
   }

   do{
    par{
     ReadOperandsPiped(opOffset);
     doneReadOp = 0;
    }
    par{
     delayStage1Loop = 0;
     doneReadOp = 1;
    }
    //don't go to next iteration
    //until other loops are finished
    while(iterationComplete){ delay; }

   }while(runReadOpLoop);
```

```
seq{
 while(delayStage1Loop){
  delay;
 }

 do{
  if(delayStage1Loop){
   delay;
  } else {
   par{
    HLatMulM5Wfv_Stage1();
    rSite_Stage1 = rSite;
    stage2Active = 1;
    runStage1Loop = runReadOpLoop;
    doneStage1 = 0;
   }
   par{
    delayStage2Loop = 0;
    doneStage1 = 1;
   }
  }
  //don't go to next iteration until other loops are finished
  while(iterationComplete){ delay; }

 }while(runStage1Loop);

}

seq{
 do{
  if(delayStage2Loop){
   delay;
  } else {
   par{
    HLatMulM5Wfv_Stage2();
    rSite_Stage2 = rSite_Stage1;
    stage3Active = 1;
    runStage2Loop = runStage1Loop;
    doneStage2 = 0;
   }
   par{
    delayStage3Loop = 0;
    doneStage2 = 1;
   }
  }

  //don't go to next iteration until other loops are finished
  while(iterationComplete){ delay; }
 }while(runStage2Loop);
}

seq{
 do{
  if(delayStage3Loop){
   delay;
  } else {
   par{
    rSite_Stage3 = rSite_Stage2;
    doneStage3 = 0;
```

```
      runStage3Loop = runStage2Loop;
      stage3_running = 1;
     }
     HLatMulM5Wfv_Stage3(resOffset);
     par{
      doneStage3 = 1;
      stage3_running = 0;
     }
    }
    //don't go to next iteration until other loops are finished
    while(iterationComplete){ delay; }

   }while(runStage3Loop);

  }
 }

 //Make sure that loop will run next time!
 runMainLoop = 1;
}
#endif

unsigned 1 runConj;
unsigned 15 CGIterations;

macro proc ConjugateGradient(){

 par{
  runConj = 1;
  CGIterations = 0;
 }

 res_old = OptimisedLatDotWfv(R_OFFSET);

 do{   //while(runConj){
  LatMulM5Wfv_Piped(TMP1_OFFSET, P_OFFSET);
  LatMulM5Wfv_Piped(TMP2_OFFSET, TMP1_OFFSET);

  dot_res = OptimisedLatDotWfv(TMP1_OFFSET);

  ldivide1(res_old, dot_res, &alpha);
  alpha_minus = FLIP(alpha);

  OptimisedLatAddSclWfvWfv(TMP2_OFFSET, R_OFFSET, 0, alpha_minus);
  OptimisedLatAddSclWfvWfv(P_OFFSET, X_OFFSET, 0, alpha);

  res_new = OptimisedLatDotWfv(R_OFFSET);

  ldivide1(res_new, res_old, &beta);

  OptimisedLatAddSclWfvWfv(R_OFFSET, P_OFFSET, 1, beta);

  res_old = res_new;

  ls1(res_old, threshold);
  delay;delay;delay;delay;delay;delay;delay;delay;
  checkConjCond = las_res1;

  if( (checkConjCond[31] == 1) || (threshold == ZERO) ){
   runConj = 0;
  } else {
   CGIterations++;
  }
 }while(runConj);
}

void usermain()
{
 unsigned 32 datatemp, datatemp2;
 f_real product;

 unsigned 20 s, t;

 unsigned 4 indexR, indexI;
 ram f_real divOut[50] with {block = "BlockRAM"};
 unsigned 6 divIndex;

 #ifdef SIMULATE
  chanout unsigned 64 gOut with {base = 16, outfile = "gOut.txt"};
 #endif

 #ifndef SIMULATE
  readStatus(datatemp);
 #else
  ReadInDataToSRAM();
  datatemp = 1;
 #endif

 GetParameters();

 #ifdef PERF_COUNTERS
  ZeroPerfCounters();
 #endif

 ConjugateGradient();

 WriteParameters();

 #ifndef SIMULATE
  writeStatus(datatemp);
 #else
  WriteOutResult();
 #endif
}

void main(void){
 delay;
 delay;
 delay;
 par{
  usermain();
#ifndef SIMULATE
  SRAMcontroller();
  sram0();
  sram1();
  sram2();
  sram3();
  sram4();
  sram5();
#endif
 }
 delay;
}
```

D.2 Gamma.hcc

```
macro expr FLIP(a) = (a ^ (1<<63));

#define EXPAND_ONE_A_S(r, p, l)
 r ## l ## p

#define EXPAND_A_S(r, p)
EXPAND_ONE_A_S(r, p, .HH) , EXPAND_ONE_A_S(r, p, .HL) ,
EXPAND_ONE_A_S(r, p, .LH) , EXPAND_ONE_A_S(r, p, .LL)

macro proc a_s_Read_Stage1(HH, HL, LH, LL, regStruct, index){
 par{
  regStruct.HH = HH[index];
  regStruct.HL = HL[index];
  regStruct.LH = LH[index];
  regStruct.LL = LL[index];
 }
}

macro expr a_s_Read_Stage2(regStruct) =
regStruct.HH @ regStruct.HL @
regStruct.LH @ regStruct.LL;

macro proc a_s_Write(HH, HL, LH, LL, data, index){
 par{
  HH[index] = data[63:48];
  HL[index] = data[47:32];
  LH[index] = data[31:16];
  LL[index] = data[15:0];
 }
}

void RetrieveSD(f_real *rr, f_real *ir, unsigned 1
input_registered){
 unsigned 4 retrieveSDX;
 unsigned 1 doRetrieveSDX;

 par{
  retrieveSDX = 0;
  doRetrieveSDX = 1;

  if(input_registered){
   delay;
   delay;
  } else {delay;}
 }


 do{
  par{
   if(retrieveSDX == 11){
    doRetrieveSDX = 0;
   } else {
    retrieveSDX++;
   }
   rr[retrieveSDX] = las_res9;
   ir[retrieveSDX] = las_res10;
  }
 }while(doRetrieveSDX);
}

unsigned 4 IssueSDX;
unsigned 1 doIssueSDX, IssueSDAdd, IssueSDRet;

macro proc IssueSD(ra, rb, ia, ib){

 par{
  IssueSDX=0;
  doIssueSDX = 1;
  IssueSDAdd = 1;
  IssueSDRet = 1;
 }

 do{
  par{
   IssueSDX++;

   if(IssueSDX == 11){
    doIssueSDX = 0;
   } else {delay;}

   la9(ra[IssueSDX], rb[IssueSDX]);
   la10(ia[IssueSDX], ib[IssueSDX]);
  }
 }while(doIssueSDX);
}

macro proc IssueSD_Registered(ra, raReg, rb, rbReg, ia, iaReg, ib,
ibReg){

 par{
  IssueSDX=0;
  doIssueSDX = 1;
  IssueSDAdd = 1;
  IssueSDRet = 1;
 }

 do{
  par{
   IssueSDX++;

   doIssueSDX = IssueSDRet;

   if(IssueSDX == 11){
    IssueSDRet = 0;
   } else {delay;}

   if(IssueSDRet){
    par{
     raReg = ra[IssueSDX];
     rbReg = rb[IssueSDX];
     iaReg = ia[IssueSDX];
     ibReg = ib[IssueSDX];
    }
   } else {delay;}
   //always issue adds, only the
//relevent results will be collected
   la9(raReg, rbReg);
   la10(iaReg, ibReg);
  }
 }while(doIssueSDX);
}

macro proc IssueMulSclSub(rmul, imul, rSub, iSub, k){
 unsigned 1 issueMulSclSub_MainLoop;
unsigned 1 issueMulSclSub_Sub;
uuunsigned 1 issueMulSclSub_Mul;
 unsigned 5 issueMulSclSub_Count;
 unsigned 4 issueMulSclSub_MulCount;
unsigned 4 issueMulSclSub_SubCount;
 unsigned 1 cycle1, cycle2, cycle3, cycle4;
unsigned 1 cycle5, cycle6, cycle7;

 //Delay 7 cycles so it procedure can be called
//in parallel with 7th add issue
 //procedure will then pick up add results and
 //issue to the multiplier on the correct cycle
 par{
  issueMulSclSub_MainLoop = 1;
  issueMulSclSub_Sub = 0;
  issueMulSclSub_Mul = 1;
  issueMulSclSub_Count = 0;
  issueMulSclSub_MulCount = 0;
  issueMulSclSub_SubCount = 0;
 }

 do{  //while(issueMulSclSub_MainLoop)
  par{
   cycle1 = issueMulSclSub_Mul;
   cycle2 = cycle1;
   cycle3 = cycle2;
   cycle4 = cycle3;
   cycle5 = cycle4;
   cycle6 = cycle5;
   cycle7 = cycle6;
   issueMulSclSub_Sub = cycle7;

   if(issueMulSclSub_MulCount == 11){
    issueMulSclSub_Mul = 0;
   } else {
    delay;
   }

   if(issueMulSclSub_Mul){
    par{
     lmul7(rmul[issueMulSclSub_MulCount], k);
     lmul8(imul[issueMulSclSub_MulCount], k);
     issueMulSclSub_MulCount++;
    }
   } else {
    delay;
   }
   //Issue the results to the subtractor
   //Subtract from the existing values
//at the point (yLat[8])
   if(issueMulSclSub_Sub){
    par{
     ls9(rSub[issueMulSclSub_SubCount], lm_res7);
     ls10(iSub[issueMulSclSub_SubCount], lm_res8);
     issueMulSclSub_SubCount++;
    }
   } else {
    delay;
   }
  }
 //run loop while either muls or sub are issuing
 } while(issueMulSclSub_Mul | issueMulSclSub_Sub);
}


unsigned 4 retMulScl_Count;
unsigned 1 retMulScl_MainLoop;
unsigned 4 retMulScl_Index;

macro proc RetrieveMulSclSubSD(rOut, iOut){
 par{
  retMulScl_Count = 0;
  retMulScl_MainLoop = 1;
 }

 do{
  par{
   rOut[retMulScl_Count] = las_res9;
   iOut[retMulScl_Count] = las_res10;

   if(retMulScl_Count == 11){
    retMulScl_MainLoop = 0;
   } else {
    par{
     retMulScl_Count++;
    }
   }
  }
 }while(retMulScl_MainLoop);
}

unsigned 4 hg5c;
f_real hg5Temp;
unsigned 1 runHG5Loop, hg5FirstHalf;

macro proc G5(rr, ir, ra, ia){
 par{
  runHG5Loop = 1;
  hg5FirstHalf = 0;
  hg5c = 0;
 }

 do{
  if(hg5FirstHalf == 0){
   par{
    rr[hg5c] = ra[hg5c];
    ir[hg5c] = ia[hg5c];
   }
  } else {
   hg5Temp = ra[hg5c];
   rr[hg5c] = FLIP(hg5Temp);
   hg5Temp = ia[hg5c];
   ir[hg5c] = FLIP(hg5Temp);
  }

  par{
   if(hg5c == 5){
    hg5FirstHalf = 1;
   } else {delay;}
   if(hg5c == 11){
    runHG5Loop = 0;
   } else {delay;}
   hg5c++;
  }
 } while(runHG5Loop);
}

#if 1

unsigned 8 cycles;

//Control Registers - control issuing and retrieval inside control
blocks
unsigned 2 controlGamma, controlGammaStore, controlGammaFlipStore;
unsigned 2 controlGammaIssue;
unsigned 2 controlGammaRetrieve;
unsigned 2 controlGLatRetrieve;
unsigned 2 control_GLat1_Mul_Issue;
unsigned 2 control_GLat1_IRAdd_Issue;
unsigned 2 control_GLat1_IRAdd_Retrieve;
unsigned 1 control_GLat1_AccAddS1_Issue;
unsigned 1 control_GLat1_AccAddS1_AddZero;
unsigned 2 control_GLat1_Acc_Retrieve;


//Run registers - control when control blocks run
unsigned 1 run_Main_Gamma_Loop, run_Gamma_Control_Loop;
unsigned 1 run_Gamma_Op_Retrieve;
unsigned 1 run_Gamma_Op_Issue;
unsigned 1 run_Gamma_Res_Retrieve;
unsigned 1 run_Gamma_Res_Store;
unsigned 1 run_Gamma_Flip_Store;

unsigned 1 run_Mul_Op_Retrieve;
unsigned 1 run_Mul_Op_Issue;
unsigned 1 run_IRAdd_Issue;
unsigned 1 run_Acc_Issue;
unsigned 1 run_Acc_Retrieve;


/*
Registers to allow memory pipelining
transformations to work.
Need a seperate registers for reading
and writing to arrays
```

```
Naming Convention:
Single Port <<array name>>_<<Direction>>
Multi Port <<array name>>_<<Port Name>>_<<Direction>>
*/

//Registers for G5pG5Gx

f_real flipped_pGx;
f_real temp_GLat1_IRAdd;

//Registers for G5mG5Gx

f_real flipped_mGx;
f_real temp_GLat0_IRAdd;


//Registers for G5pG5Gy
f_real flipped_pGy;
f_real temp_GLat3_IRAdd;


//Registers for G5mG5Gy

f_real flipped_mGy;
f_real temp_GLat2_IRAdd;


//Registers for G5pG5Gz
f_real flipped_pGz;
f_real temp_GLat5_IRAdd;


//Registers for G5mG5Gz
f_real flipped_mGz;
f_real temp_GLat4_IRAdd;

//Registers for G5pG5Gt

f_real flipped_pGt;
f_real temp_GLat7_IRAdd;

//Registers for G5mG5Gt

f_real flipped_mGt;
f_real temp_GLat6_IRAdd;

unsigned 4 zeroIndex_pGx, oneIndex_pGx, twoIndex_pGx;
unsigned 4 threeIndex_pGx;
unsigned 4 zeroIndexStore_pGx, oneIndexStore_pGx, twoIndexStore_pGx,
threeIndexStore_pGx;
unsigned 4 twoIndexStore_pGy, threeIndexStore_pGy;

unsigned 2 c_gLat1, d_gLat1;


unsigned 4 ga_smx_Index_Ret, ga_smx_Index_Store;

f_real la1_Res_For_Acc;
```

```
void GammaFunc(unsigned 1 highSpace)
{
 /*~~*/


 par
 {
  //c = 0;
  cycles = 0;
  run_Main_Gamma_Loop = 1;
  run_Gamma_Control_Loop = 1;

  zeroIndex_pGx = 0;
  oneIndex_pGx = 3;
  twoIndex_pGx = 6;
  threeIndex_pGx = 9;

  controlGamma = 0;
  controlGammaIssue = 0;
  controlGammaRetrieve = 0;
  controlGammaStore = 0;
  controlGammaFlipStore = 0;

  zeroIndexStore_pGx = 0;
  oneIndexStore_pGx = 3;
  twoIndexStore_pGx = 6;
  threeIndexStore_pGx = 9;

  twoIndexStore_pGy = 6;
  threeIndexStore_pGy = 9;

  controlGLatRetrieve = 0;

  if(highSpace == 0){
   gLat1_Index = 0;
   gLat0_Index = 0;
  } else {
   gLat1_Index = 9;
   gLat0_Index = 9;
  }

  a_smx_Index = 0;
  ga_smx_Index = 0;


  a_spx_Index = 0;
  ga_spx_Index = 0;

  c_gLat1 = 0;
  d_gLat1 = 0;

  control_GLat1_Mul_Issue = 0;

  control_GLat1_IRAdd_Issue = 0;

  control_GLat1_IRAdd_Retrieve = 0;

  control_GLat1_AccAddS1_Issue = 0;
  control_GLat1_AccAddS1_AddZero = 0;

  ga_smx_Index_Ret = 0;
  ga_smx_Index_Store = 0;

  control_GLat1_Acc_Retrieve = 0;
  ga_smx_Index_Store = 0;

  run_Gamma_Op_Retrieve = 1;
  run_Gamma_Op_Issue = 0;
  run_Gamma_Res_Retrieve = 0;
  run_Gamma_Res_Store = 0;
  run_Gamma_Flip_Store = 0;

  run_Mul_Op_Retrieve = 0;
  run_Mul_Op_Issue = 0;
  run_IRAdd_Issue = 0;
  run_Acc_Issue = 0;
  run_Acc_Retrieve = 0;
 }

 par{
  do
  //while(cycles <179)
  {
   par
   {
    cycles++;

    //Put all control register
    //updates into one loop
    //controlled by it's own
    //register

    if(cycles == 178){
     par{
      run_Main_Gamma_Loop = 0;
      run_Gamma_Control_Loop = 0;
     }
    } else {delay;}

    /*
    Retrieve yLat operands from RAM
    Read from RAM into a dedicated array to enable
    RAM pipelining

    Start issuing gamma adds/subs one cycle later

    Some of these retrieves are duplicated (the
    register already holds the correct value)
    These can be elimated once all the gamma/mul
    functions are integrated into one
    */

    if(cycles == 11)
    {
     run_Gamma_Op_Retrieve = 0;
    } else { delay;}
```

```
if(cycles == 6)
{
 run_Gamma_Res_Store = 1;
} else { delay;}
if(cycles == 18)
{
 run_Gamma_Res_Store = 0;
} else {delay;}

if(cycles == 7){
 run_Gamma_Flip_Store = 1;
} else {delay;}
if(cycles == 19){
 run_Gamma_Flip_Store = 0;
} else {delay;}

if(cycles == 8){
 run_Mul_Op_Retrieve = 1;
} else {delay;}
if(cycles == 152){
 run_Mul_Op_Retrieve = 0;
} else {delay;}

if(cycles == 9){
 run_Mul_Op_Issue = 1;
} else {delay;}
if(cycles == 153){
 run_Mul_Op_Issue = 0;
} else {delay;}

if(cycles == 17){
 run_IRAdd_Issue = 1;
} else {delay;}
if(cycles == 161){
 run_IRAdd_Issue = 0;
} else {delay;}

//Issue add of result of IR add to zero for
//first set of accumulates
if(cycles == 24){
 run_Acc_Issue = 1;
} else {delay;}
if(cycles == 168){
 run_Acc_Issue = 0;
} else {delay;}

/*
After cycle 49 all locations in RAM will be
zero'd by saving
IR result + zero to each location, start
issuing accumlate ops after this
*/
if(cycles == 72)
{
 control_GLat1_AccAddS1_AddZero = 1;
} else { delay; }

if(cycles == 32){
 run_Acc_Retrieve = 1;
```

```
  } else {delay;}
  if(cycles == 176){
   run_Acc_Retrieve = 0;
  } else {delay;}


 }
} while(run_Gamma_Control_Loop);

do{
 par{
  //if(cycles < 12)
  if(run_Gamma_Op_Retrieve == 1)
  {
   par
   {
    if(controlGamma == 0)
    {
     par
     {
      la1(yLat1R.read[zeroIndex_pGx],
       yLat1R.write[threeIndex_pGx]);
      ls2(yLat0R.read[zeroIndex_pGx],
       yLat0R.write[threeIndex_pGx]);
      la3(yLat3R[zeroIndex_pGx],
       yLat3I[threeIndex_pGx]);
      ls4(yLat2R[zeroIndex_pGx],
       yLat2I[threeIndex_pGx]);
      la5(yLat5R.read[zeroIndex_pGx],
       yLat5R.write[twoIndex_pGx]);
      ls6(yLat4R.read[zeroIndex_pGx],
       yLat4R.write[twoIndex_pGx]);
      la7(yLat7R[zeroIndex_pGx],
       yLat7I[twoIndex_pGx]);
      ls8(yLat6R[zeroIndex_pGx],
       yLat6I[twoIndex_pGx]);
     }
    } else { delay; }
    if(controlGamma == 1)
    {
     par
     {
      la1(yLat1I.read[zeroIndex_pGx],
       yLat1I.write[threeIndex_pGx]);
      ls2(yLat0I.read[zeroIndex_pGx],
       yLat0I.write[threeIndex_pGx]);
      ls3(yLat3I[zeroIndex_pGx],
       yLat3R[threeIndex_pGx]);
      la4(yLat2I[zeroIndex_pGx],
       yLat2R[threeIndex_pGx]);
      la5(yLat5I.read[zeroIndex_pGx],
       yLat5I.write[twoIndex_pGx]);
      ls6(yLat4I.read[zeroIndex_pGx],
       yLat4I.write[twoIndex_pGx]);
      ls7(yLat7I[zeroIndex_pGx],
       yLat7R[twoIndex_pGx]);
      la8(yLat6I[zeroIndex_pGx],
       yLat6R[twoIndex_pGx]);
     }
    } else { delay; }
    if(controlGamma == 2)
    {
     par
     {
      la1(yLat1R.read[oneIndex_pGx],
       yLat1R.write[twoIndex_pGx]);
      ls2(yLat0R.read[oneIndex_pGx],
       yLat0R.write[twoIndex_pGx]);
      ls3(yLat3R[oneIndex_pGx],
       yLat3I[twoIndex_pGx]);
      la4(yLat2R[oneIndex_pGx],
       yLat2I[twoIndex_pGx]);
      ls5(yLat5R.read[oneIndex_pGx],
       yLat5R.write[threeIndex_pGx]);
      la6(yLat4R.read[oneIndex_pGx],
       yLat4R.write[threeIndex_pGx]);
      la7(yLat7R[oneIndex_pGx],
       yLat7I[threeIndex_pGx]);
      ls8(yLat6R[oneIndex_pGx],
       yLat6I[threeIndex_pGx]);
     }
    } else { delay; }
    if(controlGamma == 3)
    {
     par
     {
      la1(yLat1I.read[oneIndex_pGx],
       yLat1I.write[twoIndex_pGx]);
      ls2(yLat0I.read[oneIndex_pGx],
       yLat0I.write[twoIndex_pGx]);
      la3(yLat3I[oneIndex_pGx],
       yLat3R[twoIndex_pGx]);
      ls4(yLat2I[oneIndex_pGx],
       yLat2R[twoIndex_pGx]);
      ls5(yLat5I.read[oneIndex_pGx],
       yLat5I.write[threeIndex_pGx]);
      la6(yLat4I.read[oneIndex_pGx],
       yLat4I.write[threeIndex_pGx]);
      ls7(yLat7I[oneIndex_pGx],
       yLat7R[threeIndex_pGx]);
      la8(yLat6I[oneIndex_pGx],
       yLat6R[threeIndex_pGx]);

      zeroIndex_pGx++;
      oneIndex_pGx++;
      twoIndex_pGx++;
      threeIndex_pGx++;
     }
    } else { delay; }

    controlGamma++;
   }
  } else { delay; }
 }

}while(run_Main_Gamma_Loop);

do{
```

```
par{

 /*
 Store Gamma addition results

 Storage of sign flipped results will need to be
 pipelined for half of the gamma functions
 because of lack of memory ports

 The other half will need to store the flipped
 results on the same cycle as the non-flipped
 because the the flipped results are stored to
 different arrays than the non flipped.
 */


 if(run_Gamma_Res_Store == 1)
 {
  par{
   controlGammaStore++;

   if(controlGammaStore == 0)
   {
    par
    {
     //a_smxR.write[zeroIndexStore_pGx] =
                    las_res1;
     a_s_Write(EXPAND_A_S(a_smxR, .write),
      las_res1, zeroIndexStore_pGx);
     a_s_Write(EXPAND_A_S(a_spxR, .write),
      las_res2, zeroIndexStore_pGx);
     a_s_Write(EXPAND_A_S(a_smyR, .write),
      las_res3, zeroIndexStore_pGx);
     a_s_Write(EXPAND_A_S(a_spyR, .write),
      las_res4, zeroIndexStore_pGx);
     a_s_Write(EXPAND_A_S(a_smzR, .write),
      las_res5, zeroIndexStore_pGx);
     a_s_Write(EXPAND_A_S(a_spzR, .write),
      las_res6, zeroIndexStore_pGx);
     a_s_Write(EXPAND_A_S(a_smtR, .write),
      las_res7, zeroIndexStore_pGx);
     a_s_Write(EXPAND_A_S(a_sptR, .write),
      las_res8, zeroIndexStore_pGx);

     //Store flipped results that don't need
     //to be delayed
     a_smxR_Write_In = FLIP(las_res1);
     a_spxR_Write_In = las_res2;
     a_s_Write(EXPAND_A_S(a_smyI, .write),
      FLIP(las_res3), threeIndexStore_pGy);
     a_s_Write(EXPAND_A_S(a_spyI, .write),
      las_res4, threeIndexStore_pGy);
     a_smzR_Write_In = FLIP(las_res5);
     a_spzR_Write_In = las_res6;
     a_s_Write(EXPAND_A_S(a_smtI, .write),
       FLIP(las_res7), twoIndexStore_pGy);
     a_s_Write(EXPAND_A_S(a_sptI, .write),
      las_res8, twoIndexStore_pGy);
    }
```

```
} else { delay; }

if(controlGammaStore == 1)
{
 par
 {
  a_s_Write(EXPAND_A_S(a_smxI, .write),
   las_res1, zeroIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_spxI, .write),
   las_res2, zeroIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_smyI, .write),
   las_res3, zeroIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_spyI, .write),
   las_res4, zeroIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_smzI, .write),
   las_res5, zeroIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_spzI, .write),
   las_res6, zeroIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_smtI, .write),
   las_res7, zeroIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_sptI, .write),
   las_res8, zeroIndexStore_pGx);

  //Store flipped results that don't need
  //to be delayed
  a_smxI_Write_In = FLIP(las_res1);
  a_spxI_Write_In = las_res2;
  a_s_Write(EXPAND_A_S(a_smyR, .write),
   las_res3, threeIndexStore_pGy);
  a_s_Write(EXPAND_A_S(a_spyR, .write),
   FLIP(las_res4), threeIndexStore_pGy);
  a_smzI_Write_In = FLIP(las_res5);
  a_spzI_Write_In = las_res6;
  a_s_Write(EXPAND_A_S(a_smtR, .write),
   las_res7, twoIndexStore_pGy);

  a_s_Write(EXPAND_A_S(a_sptR, .write),
   FLIP(las_res8), twoIndexStore_pGy);
 }
} else { delay; }

if(controlGammaStore == 2)
{
 par{
  a_s_Write(EXPAND_A_S(a_smxR, .write),
   las_res1, oneIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_spxR, .write),
   las_res2, oneIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_smyR, .write),
   las_res3, oneIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_spyR, .write),
   las_res4, oneIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_smzR, .write),
   las_res5, oneIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_spzR, .write),
   las_res6, oneIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_smtR, .write),
   las_res7, oneIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_sptR, .write),
   las_res8, oneIndexStore_pGx);

  //Store flipped results that don't need
  //to be delayed
  a_smxR_Write_In = FLIP(las_res1);
  a_spxR_Write_In = las_res2;
  a_s_Write(EXPAND_A_S(a_smyI, .write),
   las_res3, twoIndexStore_pGy);
  a_s_Write(EXPAND_A_S(a_spyI, .write),
   FLIP(las_res4), twoIndexStore_pGy);
  a_smzR_Write_In = las_res5;
  a_spzR_Write_In = FLIP(las_res6);
  a_s_Write(EXPAND_A_S(a_smtI, .write),
   FLIP(las_res7), threeIndexStore_pGy);
  a_s_Write(EXPAND_A_S(a_sptI, .write),
   las_res8, threeIndexStore_pGy);
 }
} else { delay; }

if(controlGammaStore == 3)
{
 par
 {
  a_s_Write(EXPAND_A_S(a_smxI, .write),
   las_res1, oneIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_spxI, .write),
   las_res2, oneIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_smyI, .write),
   las_res3, oneIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_spyI, .write),
   las_res4, oneIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_smzI, .write),
   las_res5, oneIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_spzI, .write),
   las_res6, oneIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_smtI, .write),
   las_res7, oneIndexStore_pGx);
  a_s_Write(EXPAND_A_S(a_sptI, .write),
   las_res8, oneIndexStore_pGx);

  //Store flipped results that don't need
  //to be delayed
  a_smxI_Write_In = FLIP(las_res1);
  a_spxI_Write_In = las_res2;

  a_s_Write(EXPAND_A_S(a_smyR, .write),
   FLIP(las_res3), twoIndexStore_pGy);
  a_s_Write(EXPAND_A_S(a_spyR, .write),
   las_res4, twoIndexStore_pGy);
  a_smzI_Write_In = las_res5;
  a_spzI_Write_In = FLIP(las_res6);
  a_s_Write(EXPAND_A_S(a_smtR, .write),
   las_res7, threeIndexStore_pGy);
  a_s_Write(EXPAND_A_S(a_sptR, .write),
   FLIP(las_res8), threeIndexStore_pGy);

  zeroIndexStore_pGx++;
  oneIndexStore_pGx++;
  twoIndexStore_pGy++;
```

```
      threeIndexStore_pGy++;
     }
    } else { delay; }
   }
  } else { delay; }
 }
}while(run_Main_Gamma_Loop);

do{
 par{
  //Finish pipelining of flipped results
  if(cycles == 19)
  {
   par{
    a_smxI_Write_In = FLIP(las_res1);
    a_spxI_Write_In = las_res2;
    a_smzI_Write_In = FLIP(las_res5);
    a_spzI_Write_In = las_res6;
   }
  } else { delay; }
 }
}while(run_Main_Gamma_Loop);

do{
 par{
  /*
  Store the sign flipped gamma results

  For some gamma functions these have been
  carried over from earlier cycles
  */

  if(run_Gamma_Flip_Store == 1)
  {
   par{
    controlGammaFlipStore++;

    if(controlGammaFlipStore == 0)
    {
     par{
      a_s_Write(EXPAND_A_S(a_smxR, .write),
       a_smxR_Write_In,
       threeIndexStore_pGx);
      a_s_Write(EXPAND_A_S(a_spxR, .write),
        a_spxR_Write_In,
       threeIndexStore_pGx);
      a_s_Write(EXPAND_A_S(a_smzR, .write),
       a_smzR_Write_In, twoIndexStore_pGx);
      a_s_Write(EXPAND_A_S(a_spzR, .write),
       a_spzR_Write_In, twoIndexStore_pGx);
     }
    } else { delay; }

    if(controlGammaFlipStore == 1)
    {
     par
     {
      a_s_Write(EXPAND_A_S(a_smxI, .write),
        a_smxI_Write_In,
       threeIndexStore_pGx);
      a_s_Write(EXPAND_A_S(a_spxI, .write),
       a_spxI_Write_In,
       threeIndexStore_pGx);
      a_s_Write(EXPAND_A_S(a_smzI, .write),
       a_smzI_Write_In, twoIndexStore_pGx);
      a_s_Write(EXPAND_A_S(a_spzI, .write),
       a_spzI_Write_In, twoIndexStore_pGx);
     }
    } else { delay; }

    if(controlGammaFlipStore == 2)
    {
     par{
      a_s_Write(EXPAND_A_S(a_smxR, .write),
       a_smxR_Write_In, twoIndexStore_pGx);
      a_s_Write(EXPAND_A_S(a_spxR, .write),
       a_spxR_Write_In, twoIndexStore_pGx);
      a_s_Write(EXPAND_A_S(a_smzR, .write),
       a_smzR_Write_In,
       threeIndexStore_pGx);
      a_s_Write(EXPAND_A_S(a_spzR, .write),
       a_spzR_Write_In,
       threeIndexStore_pGx);
     }
    } else { delay; }

    if(controlGammaFlipStore == 3)
    {
     par
     {
      a_s_Write(EXPAND_A_S(a_smxI, .write),
       a_smxI_Write_In, twoIndexStore_pGx);
      a_s_Write(EXPAND_A_S(a_spxI, .write),
       a_spxI_Write_In, twoIndexStore_pGx);
      a_s_Write(EXPAND_A_S(a_smzI, .write),
       a_smzI_Write_In,
       threeIndexStore_pGx);
      a_s_Write(EXPAND_A_S(a_spzI, .write),
       a_spzI_Write_In,
       threeIndexStore_pGx);

      threeIndexStore_pGx++;
      twoIndexStore_pGx++;
     }
    } else { delay; }
   }
  } else { delay; }
 }
}while(run_Main_Gamma_Loop);

do{
 par{
  //Mul section of pipe

  //Retrieve operands for multiply from gLat
  //First store to a_smx will be available in
  //cycle 10 (stored in 9)
```

```
if(run_Mul_Op_Retrieve == 1)
{
 par{
  controlGLatRetrieve++;

  if(controlGLatRetrieve == 0)
  {
   par
   {
    gLat1R_Out =
     gLat1R.readWrite[gLat1_Index];
    a_s_Read_Stage1(EXPAND_A_S
     (a_smxR, .readWrite),
     a_smxR_Read_Out, a_smx_Index);

    gLat0R_Out =
     gLat0R.readWrite[gLat0_Index];
    a_s_Read_Stage1(EXPAND_A_S(a_spxR,
     .readWrite), a_spxR_Read_Out,
     a_spx_Index);

    gLat3R_Out =
     gLat3R.readWrite[gLat1_Index];
    a_s_Read_Stage1(EXPAND_A_S(a_smyR,
     .readWrite), a_smyR_Read_Out,
     a_smx_Index);

    gLat2R_Out =
     gLat2R.readWrite[gLat0_Index];
    a_s_Read_Stage1(EXPAND_A_S(a_spyR,
     .readWrite), a_spyR_Read_Out,
     a_spx_Index);

    gLat5R_Out =
     gLat5R.readWrite[gLat1_Index];
    a_s_Read_Stage1(EXPAND_A_S(a_smzR,
     .readWrite), a_smzR_Read_Out,
     a_smx_Index);

    gLat4R_Out =
     gLat4R.readWrite[gLat0_Index];
    a_s_Read_Stage1(EXPAND_A_S(a_spzR,
     .readWrite), a_spzR_Read_Out,
     a_spx_Index);

    gLat7R_Out =
     gLat7R.readWrite[gLat1_Index];
    a_s_Read_Stage1(EXPAND_A_S(a_smtR,
     .readWrite), a_smtR_Read_Out,
     a_smx_Index);

    gLat6R_Out =
     gLat6R.readWrite[gLat0_Index];
    a_s_Read_Stage1(EXPAND_A_S(a_sptR,
     .readWrite), a_sptR_Read_Out,
     a_spx_Index);
   }
  } else { delay; }
```

```
if(controlGLatRetrieve == 1)
{
 par
 {
  gLat1I_Out =
   gLat1I.readWrite[gLat1_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_smxI,
   .readWrite), a_smxI_Read_Out,
   a_smx_Index);

  gLat0I_Out =
   gLat0I.readWrite[gLat0_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_spxI,
   .readWrite), a_spxI_Read_Out,
   a_spx_Index);

  gLat3I_Out =
   gLat3I.readWrite[gLat1_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_smyI,
   .readWrite), a_smyI_Read_Out,
   a_smx_Index);

  gLat2I_Out =
   gLat2I.readWrite[gLat0_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_spyI,
   .readWrite), a_spyI_Read_Out,
   a_spx_Index);

  gLat5I_Out =
   gLat5I.readWrite[gLat1_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_smzI,
   .readWrite), a_smzI_Read_Out,
   a_smx_Index);

  gLat4I_Out =
   gLat4I.readWrite[gLat0_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_spzI,
   .readWrite), a_spzI_Read_Out,
   a_spx_Index);

  gLat7I_Out =
   gLat7I.readWrite[gLat1_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_smtI,
   .readWrite), a_smtI_Read_Out,
   a_smx_Index);

  gLat6I_Out =
   gLat6I.readWrite[gLat0_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_sptI,
   .readWrite), a_sptI_Read_Out,
   a_spx_Index);
 }
} else { delay; }

if(controlGLatRetrieve == 2)
{
 par
 {
  gLat1R_Out =
   gLat1R.readWrite[gLat1_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_smxI,
   .readWrite), a_smxI_Read_Out,
   a_smx_Index);

  gLat0R_Out =
   gLat0R.readWrite[gLat0_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_spxI,
   .readWrite), a_spxI_Read_Out,
   a_spx_Index);

  gLat3R_Out =
   gLat3R.readWrite[gLat1_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_smyI,
   .readWrite), a_smyI_Read_Out,
   a_smx_Index);

  gLat2R_Out =
   gLat2R.readWrite[gLat0_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_spyI,
   .readWrite), a_spyI_Read_Out,
   a_spx_Index);

  gLat5R_Out =
   gLat5R.readWrite[gLat1_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_smzR,
   .readWrite), a_smzR_Read_Out,
   a_smx_Index);

  gLat4R_Out =
   gLat4R.readWrite[gLat0_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_spzI,
   .readWrite), a_spzI_Read_Out,
   a_spx_Index);

  gLat7R_Out =
   gLat7R.readWrite[gLat1_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_smtR,
   .readWrite), a_smtR_Read_Out,
   a_smx_Index);

  gLat6R_Out =
   gLat6R.readWrite[gLat0_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_sptI,
   .readWrite), a_sptI_Read_Out,
   a_spx_Index);
 }
} else { delay; }

if(controlGLatRetrieve == 3)
{
 par
 {
  gLat1I_Out =
   gLat1I.readWrite[gLat1_Index];
  a_s_Read_Stage1(EXPAND_A_S(a_smxR,
   .readWrite), a_smxR_Read_Out,
   a_smx_Index);
```

```
gLat0I_Out =
 gLat0I.readWrite[gLat0_Index];
a_s_Read_Stage1(EXPAND_A_S(a_spxR,
 .readWrite), a_spxR_Read_Out,
 a_spx_Index);

gLat3I_Out =
 gLat3I.readWrite[gLat1_Index];
a_s_Read_Stage1(EXPAND_A_S(a_smyR,
 .readWrite), a_smyR_Read_Out,
 a_smx_Index);

gLat2I_Out =
 gLat2I.readWrite[gLat0_Index];
a_s_Read_Stage1(EXPAND_A_S(a_spyR,
 .readWrite), a_spyR_Read_Out,
 a_spx_Index);

gLat5I_Out =
 gLat5I.readWrite[gLat1_Index];
a_s_Read_Stage1(EXPAND_A_S(a_smzR,
 .readWrite), a_smzR_Read_Out,
 a_smx_Index);

gLat4I_Out =
 gLat4I.readWrite[gLat0_Index];
a_s_Read_Stage1(EXPAND_A_S(a_spzR,
 .readWrite), a_spzR_Read_Out,
 a_spx_Index);

gLat7I_Out =
 gLat7I.readWrite[gLat1_Index];
a_s_Read_Stage1(EXPAND_A_S(a_smtR,
 .readWrite), a_smtR_Read_Out,
 a_smx_Index);

gLat6I_Out =
 gLat6I.readWrite[gLat0_Index];
a_s_Read_Stage1(EXPAND_A_S(a_sptR,
 .readWrite), a_sptR_Read_Out,
 a_spx_Index);

if ((d_gLat1 == 3) && (c_gLat1 == 2)){
 par{
  d_gLat1=0;
  c_gLat1=0;
  gLat1_Index++;
  gLat0_Index -= 5;
  a_smx_Index -= 8;
  a_spx_Index -= 8;
 }
} else if(c_gLat1==2){
 par{
  c_gLat1=0;
  d_gLat1++;
  a_smx_Index += 3;
  a_spx_Index += 3;
  gLat1_Index -= 2;
  gLat0_Index -= 6;
```

```
       }
      } else {
       par{
        c_gLat1++;
        gLat1_Index++;
        gLat0_Index += 3;
       }
      }
     }
    } else { delay; }
   }
  } else { delay; }
 }
}while(run_Main_Gamma_Loop);

do{
 par{
  //Issue retrieved operands to multiplier unit

  if(run_Mul_Op_Issue == 1)
  {
   par
   {
    control_GLat1_Mul_Issue++;

    if(control_GLat1_Mul_Issue == 0)
    {
     par{
      lmul1(gLat1R_Out,
      a_s_Read_Stage2(a_smxR_Read_Out));
      lmul2(gLat0R_Out,
      a_s_Read_Stage2(a_spxR_Read_Out));
      lmul3(gLat3R_Out,
      a_s_Read_Stage2(a_smyR_Read_Out));
      lmul4(gLat2R_Out,
      a_s_Read_Stage2(a_spyR_Read_Out));
      lmul5(gLat5R_Out,
      a_s_Read_Stage2(a_smzR_Read_Out));
      lmul6(gLat4R_Out,
      a_s_Read_Stage2(a_spzR_Read_Out));
      lmul7(gLat7R_Out,
      a_s_Read_Stage2(a_smtR_Read_Out));
      lmul8(gLat6R_Out,
      a_s_Read_Stage2(a_sptR_Read_Out));
     }
     } else { delay; }

    if(control_GLat1_Mul_Issue == 1)
    {
     par{
      lmul1(gLat1I_Out,
      a_s_Read_Stage2(a_smxI_Read_Out));
      lmul2(gLat0I_Out,
      a_s_Read_Stage2(a_spxI_Read_Out));
      lmul3(gLat3I_Out,
      a_s_Read_Stage2(a_smyI_Read_Out));
      lmul4(gLat2I_Out,
      a_s_Read_Stage2(a_spyI_Read_Out));
      lmul5(gLat5I_Out,
      a_s_Read_Stage2(a_smzI_Read_Out));
      lmul6(gLat4I_Out,
      a_s_Read_Stage2(a_spzI_Read_Out));
      lmul7(gLat7I_Out,
      a_s_Read_Stage2(a_smtI_Read_Out));
      lmul8(gLat6I_Out,
      a_s_Read_Stage2(a_sptI_Read_Out));
     }
    } else { delay;}

    if(control_GLat1_Mul_Issue == 2)
    {
     par{
      lmul1(gLat1R_Out,
      a_s_Read_Stage2(a_smxI_Read_Out));
      lmul2(gLat0R_Out,
      a_s_Read_Stage2(a_spxI_Read_Out));
      lmul3(gLat3R_Out,
      a_s_Read_Stage2(a_smyI_Read_Out));
      lmul4(gLat2R_Out,
      a_s_Read_Stage2(a_spyI_Read_Out));
      lmul5(gLat5R_Out,
      a_s_Read_Stage2(a_smzI_Read_Out));
      lmul6(gLat4R_Out,
      a_s_Read_Stage2(a_spzI_Read_Out));
      lmul7(gLat7R_Out,
      a_s_Read_Stage2(a_smtI_Read_Out));
      lmul8(gLat6R_Out,
      a_s_Read_Stage2(a_sptI_Read_Out));
     }
    } else { delay; }

    if(control_GLat1_Mul_Issue == 3)
    {
     par{
      lmul1(gLat1I_Out,
      a_s_Read_Stage2(a_smxR_Read_Out));
      lmul2(gLat0I_Out,
      a_s_Read_Stage2(a_spxR_Read_Out));
      lmul3(gLat3I_Out,
      a_s_Read_Stage2(a_smyR_Read_Out));
      lmul4(gLat2I_Out,
      a_s_Read_Stage2(a_spyR_Read_Out));
      lmul5(gLat5I_Out,
      a_s_Read_Stage2(a_smzR_Read_Out));
      lmul6(gLat4I_Out,
      a_s_Read_Stage2(a_spzR_Read_Out));
      lmul7(gLat7I_Out,
      a_s_Read_Stage2(a_smtR_Read_Out));
      lmul8(gLat6I_Out,
      a_s_Read_Stage2(a_sptR_Read_Out));
     }
    } else { delay; }
   }
  } else { delay; }
 }
}while(run_Main_Gamma_Loop);

do{
   par{
    //Issue IR Adds on every second cycle

    if(run_IRAdd_Issue == 1)
    {
     par
     {
      control_GLat1_IRAdd_Issue++;

      if((control_GLat1_IRAdd_Issue == 0) ||
(control_GLat1_IRAdd_Issue == 2))
      {
       par{
        temp_GLat1_IRAdd = lm_res1;
        temp_GLat0_IRAdd = lm_res2;
        temp_GLat3_IRAdd = lm_res3;
        temp_GLat2_IRAdd = lm_res4;
        temp_GLat5_IRAdd = lm_res5;
        temp_GLat4_IRAdd = lm_res6;
        temp_GLat7_IRAdd = lm_res7;
        temp_GLat6_IRAdd = lm_res8;
       }
      } else { delay; }

      if(control_GLat1_IRAdd_Issue == 1)
      {
       par{
        la1(temp_GLat1_IRAdd, lm_res1);
        ls2(temp_GLat0_IRAdd, lm_res2);
        la3(temp_GLat3_IRAdd, lm_res3);
        ls4(temp_GLat2_IRAdd, lm_res4);
        la5(temp_GLat5_IRAdd, lm_res5);
        ls6(temp_GLat4_IRAdd, lm_res6);
        la7(temp_GLat7_IRAdd, lm_res7);
        ls8(temp_GLat6_IRAdd, lm_res8);
       }
      } else {delay;}

      if(control_GLat1_IRAdd_Issue == 3)
      {
       par{
        ls1(temp_GLat1_IRAdd, lm_res1);
        la2(temp_GLat0_IRAdd, lm_res2);
        ls3(temp_GLat3_IRAdd, lm_res3);
        la4(temp_GLat2_IRAdd, lm_res4);
        ls5(temp_GLat5_IRAdd, lm_res5);
        la6(temp_GLat4_IRAdd, lm_res6);
        ls7(temp_GLat7_IRAdd, lm_res7);
        la8(temp_GLat6_IRAdd, lm_res8);
       }
      } else {delay;}
     }
    } else { delay; }
   }
  }while(run_Main_Gamma_Loop);

  do{
   par{
```

```
if(run_Acc_Issue == 1)
{
 par
 {
  control_GLat1_IRAdd_Retrieve++;

  /*
  Add IR result to zero for the 1st 24
  accumulates
  (avoids the need for a seperate loop to
  zero the result RAM
  */
  if((control_GLat1_IRAdd_Retrieve[0] == 1)
   && (control_GLat1_AccAddS1_AddZero == 0))
  {
   par{
    la1(0, las_res1);
    la2(0, las_res2);
    la3(0, las_res3);
    la4(0, las_res4);
    la5(0, las_res5);
    la6(0, las_res6);
    la7(0, las_res7);
    la8(0, las_res8);
   }
  } else { delay; }

  // Controls issuing of accumulates
  if(control_GLat1_AccAddS1_AddZero == 1)
  {
   par
   {
    //Retrieve ga_spxR value for accumulate
    if(control_GLat1_IRAdd_Retrieve == 0)
    {
     par{
      ga_smxR_Read_Out =
       ga_smxR.read[ga_smx_Index_Ret];
      ga_spxR_Read_Out =
       ga_spxR.read[ga_smx_Index_Ret];
      ga_smyR_Read_Out =
       ga_smyR.read[ga_smx_Index_Ret];
      ga_spyR_Read_Out =
       ga_spyR.read[ga_smx_Index_Ret];
      ga_smzR_Read_Out =
       ga_smzR.read[ga_smx_Index_Ret];
      ga_spzR_Read_Out =
       ga_spzR.read[ga_smx_Index_Ret];
      ga_smtR_Read_Out =
       ga_smtR.read[ga_smx_Index_Ret];
      ga_sptR_Read_Out =
       ga_sptR.read[ga_smx_Index_Ret];
     }
    } else { delay; }

    if(control_GLat1_IRAdd_Retrieve == 1)
    {
     par{
      la1(ga_smxR_Read_Out, las_res1);
      la2(ga_spxR_Read_Out, las_res2);
      la3(ga_smyR_Read_Out, las_res3);
      la4(ga_spyR_Read_Out, las_res4);
      la5(ga_smzR_Read_Out, las_res5);
      la6(ga_spzR_Read_Out, las_res6);
      la7(ga_smtR_Read_Out, las_res7);
      la8(ga_sptR_Read_Out, las_res8);
     }
    } else {delay;}

    if(control_GLat1_IRAdd_Retrieve == 2)
    {
     par
     {
      ga_smxI_Read_Out =
       ga_smxI.read[ga_smx_Index_Ret];
      ga_spxI_Read_Out =
       ga_spxI.read[ga_smx_Index_Ret];
      ga_smyI_Read_Out =
       ga_smyI.read[ga_smx_Index_Ret];
      ga_spyI_Read_Out =
       ga_spyI.read[ga_smx_Index_Ret];
      ga_smzI_Read_Out =
       ga_smzI.read[ga_smx_Index_Ret];
      ga_spzI_Read_Out =
       ga_spzI.read[ga_smx_Index_Ret];
      ga_smtI_Read_Out =
       ga_smtI.read[ga_smx_Index_Ret];
      ga_sptI_Read_Out =
       ga_sptI.read[ga_smx_Index_Ret];

      if(ga_smx_Index_Ret == 11)
      {
       ga_smx_Index_Ret = 0;
      } else {
       ga_smx_Index_Ret++;
      }
     }
    } else {delay;}

    if(control_GLat1_IRAdd_Retrieve == 3)
    {
     par{
      la1(ga_smxI_Read_Out, las_res1);
      la2(ga_spxI_Read_Out, las_res2);
      la3(ga_smyI_Read_Out, las_res3);
      la4(ga_spyI_Read_Out, las_res4);
      la5(ga_smzI_Read_Out, las_res5);
      la6(ga_spzI_Read_Out, las_res6);
      la7(ga_smtI_Read_Out, las_res7);
      la8(ga_sptI_Read_Out, las_res8);
     }
    } else {delay;}
   }
  }else { delay; }

 }
} else { delay; }
```

```
   }
  }while(run_Main_Gamma_Loop);

  do{
   par{

    if(run_Acc_Retrieve == 1)
    {
     par{
      control_GLat1_Acc_Retrieve++;

      if(control_GLat1_Acc_Retrieve == 0)
      {
       par{
        ga_smxR.write[ga_smx_Index_Store] = las_res1;
        ga_spxR.write[ga_smx_Index_Store] = las_res2;
        ga_smyR.write[ga_smx_Index_Store] = las_res3;
        ga_spyR.write[ga_smx_Index_Store] = las_res4;
        ga_smzR.write[ga_smx_Index_Store] = las_res5;
        ga_spzR.write[ga_smx_Index_Store] = las_res6;
        ga_smtR.write[ga_smx_Index_Store] = las_res7;
        ga_sptR.write[ga_smx_Index_Store] = las_res8;
       }
      } else {delay;}

      if(control_GLat1_Acc_Retrieve == 2)
      {
       par{
        ga_smxI.write[ga_smx_Index_Store] = las_res1;
        ga_spxI.write[ga_smx_Index_Store] = las_res2;
        ga_smyI.write[ga_smx_Index_Store] = las_res3;
        ga_spyI.write[ga_smx_Index_Store] = las_res4;
        ga_smzI.write[ga_smx_Index_Store] = las_res5;
        ga_spzI.write[ga_smx_Index_Store] = las_res6;
        ga_smtI.write[ga_smx_Index_Store] = las_res7;
        ga_sptI.write[ga_smx_Index_Store] = las_res8;

       }
      } else {delay;}

      if(control_GLat1_Acc_Retrieve == 3)
      {
       par
       {
        if(ga_smx_Index_Store == 11)
        {
         ga_smx_Index_Store = 0;
        } else {
         ga_smx_Index_Store++;
        }

       }

      } else {delay;}
     }
    } else {delay;}
   }
  }while(run_Main_Gamma_Loop);
 }
```

```
}

#endif
/*
macro proc CopyWfvSeq(to, from){
 seq{
  to[0] = from[0];
  to[1] = from[1];
  to[2] = from[2];
  to[3] = from[3];
  to[4] = from[4];
  to[5] = from[5];
  to[6] = from[6];
  to[7] = from[7];
  to[8] = from[8];
  to[9] = from[9];
  to[10] = from[10];
  to[11] = from[11];
 }
}
*/
ram f_real ga_spzR_Stage2[12], ga_spzI_Stage2[12],
ga_smzR_Stage2[12], ga_smzI_Stage2[12];
ram f_real ga_sptR_Stage2[12], ga_sptI_Stage2[12],
ga_smtR_Stage2[12], ga_smtI_Stage2[12];

f_real ga_spzR_Stage2_Read_Out, ga_spzI_Stage2_Read_Out,
ga_smzR_Stage2_Read_Out, ga_smzI_Stage2_Read_Out;
f_real ga_sptR_Stage2_Read_Out, ga_sptI_Stage2_Read_Out,
ga_smtR_Stage2_Read_Out, ga_smtI_Stage2_Read_Out;

ram f_real g5R_Stage2[12], g5I_Stage2[12];

macro proc CopyWfvSeq(){
 unsigned 4 indexRet1, indexRet2, indexRet3, indexRet4, indexRet5;
 unsigned 4 indexStore1, indexStore2, indexStore3, indexStore4;
 unsigned 1 runRet, runStore;
 signal unsigned 4 incIndex;

 par{
  indexRet1 = 0;
  indexRet2 = 0;
  indexRet3 = 0;
  indexRet4 = 0;
  indexRet5 = 0;

  indexStore1 = 0;
  indexStore2 = 0;
  indexStore3 = 0;
  indexStore4 = 0;

  runRet = 1;
  runStore = 0;
 }

 do{
  par{
   incIndex = indexRet1 + 1;
   indexRet1 = incIndex;
```

```
    indexRet2 = incIndex;
    indexRet3 = incIndex;
    indexRet4 = incIndex;
    indexRet5 = incIndex;

    indexStore1 = indexRet1;
    indexStore2 = indexRet2;
    indexStore3 = indexRet3;
    indexStore4 = indexRet4;

    runStore = runRet;

    if(indexRet1 == 11){
     runRet = 0;
    } else {delay;}

    if(runRet){
     par{
      ga_spzR_Read_Out = ga_spzR.read[indexRet1];
      ga_smzR_Read_Out = ga_smzR.read[indexRet1];
      ga_sptR_Read_Out = ga_sptR.read[indexRet2];
      ga_smtR_Read_Out = ga_smtR.read[indexRet2];
      ga_spzI_Read_Out = ga_spzI.read[indexRet3];
      ga_smzI_Read_Out = ga_smzI.read[indexRet3];
      ga_sptI_Read_Out = ga_sptI.read[indexRet4];
      ga_smtI_Read_Out = ga_smtI.read[indexRet4];
      g5R_Stage2[indexRet5] = g5a_sR[indexRet5];
      g5I_Stage2[indexRet5] = g5a_sI[indexRet5];
     }
    } else {delay;}

    if(runStore){
     par{
      ga_spzR_Stage2[indexStore1] = ga_spzR_Read_Out;
      ga_smzR_Stage2[indexStore1] = ga_smzR_Read_Out;
      ga_sptR_Stage2[indexStore2] = ga_sptR_Read_Out;
      ga_smtR_Stage2[indexStore2] = ga_smtR_Read_Out;
      ga_spzI_Stage2[indexStore3] = ga_spzI_Read_Out;
      ga_smzI_Stage2[indexStore3] = ga_smzI_Read_Out;
      ga_sptI_Stage2[indexStore4] = ga_sptI_Read_Out;
      ga_smtI_Stage2[indexStore4] = ga_smtI_Read_Out;
     }
    } else {delay;}
   }
  } while(runRet | runStore);
}


macro proc HLatMulM5Wfv_Stage1(){
 par{
  GammaFunc(ReadGl3AltRamHalf_Stage1);
  seq{
   delay;delay;delay;delay;delay;delay;
   delay;delay;delay;delay;delay;delay;
   G5(g5a_sR, g5a_sI, yLat8R, yLat8I);
  }
 }
}
```

```
macro proc HLatMulM5Wfv_Stage2(){

 unsigned 1 startMulSclIssue;

 //Need to start retrieving the results fo the adds in the ninth
cycle after the first is issued
 //To do this, call the pairs of real & imaginary functions in
sequence
 //In parallel, wait 9 cycles then call the retrieve functions.
 par{
  startMulSclIssue = 0;
  /*
  CopyWfvSeq(ga_spzR_Stage2, ga_spzR.read);
  CopyWfvSeq(ga_smzR_Stage2, ga_smzR.read);
  CopyWfvSeq(ga_sptR_Stage2, ga_sptR.read);
  CopyWfvSeq(ga_smtR_Stage2, ga_smtR.read);
  CopyWfvSeq(ga_spzI_Stage2, ga_spzI.read);
  CopyWfvSeq(ga_smzI_Stage2, ga_smzI.read);
  CopyWfvSeq(ga_sptI_Stage2, ga_sptI.read);
  CopyWfvSeq(ga_smtI_Stage2, ga_smtI.read);*/

  CopyWfvSeq();

  //Copy results of previous stage G5 calculation
  //CopyWfvSeq(g5R_Stage2, g5a_sR);
  //CopyWfvSeq(g5I_Stage2, g5a_sI);

  //Call issue functions
  seq{
   //Add up the results of the 8 Gamma/Mul functions
   IssueSD_Registered(ga_spxR.read, ga_spxR_Read_Out,
    ga_smxR.read, ga_smxR_Read_Out, ga_spxI.read,
    ga_spxI_Read_Out, ga_smxI.read, ga_smxI_Read_Out);
   IssueSD_Registered(ga_spyR.read, ga_spyR_Read_Out,
    ga_smyR.read, ga_smyR_Read_Out, ga_spyI.read,
    ga_spyI_Read_Out, ga_smyI.read, ga_smyI_Read_Out);
   IssueSD(ga_spzR_Stage2, ga_smzR_Stage2,
    ga_spzI_Stage2, ga_smzI_Stage2);
   IssueSD(ga_sptR_Stage2, ga_smtR_Stage2,
    ga_sptI_Stage2, ga_smtI_Stage2);

   //Issue the second sets of adds, operands for these
   //adds have all been retrieved
   //The next one is complete by now, only ~9 operands
   //going into add4 are still to be calculated
   IssueSD(radd1, radd2, iadd1, iadd2);

   //This set's operands have all been retrieved, the
   //previous set of issues are in the pipe now
   IssueSD(radd3, radd4, iadd3, iadd4);

   IssueSD(radd5, radd6.read, iadd5, iadd6.read);


   delay;
   while(run_Mul_Op_Issue | !startMulSclIssue){delay;}

   IssueMulSclSub(radd7.read, iadd7.read, g5R_Stage2,
    g5I_Stage2, kappa);
  }

  seq{
   //Need to wait X cycles before beginning to retrieve
   //results from the adder pipe
   //Where x is the latency of the adder pipe
   //Currently the zeroing of the counters cancel each
   //other out between functions
   delay;
   delay;

   delay;
   delay;

   delay;
   delay;
   delay;

   //Retrieve the results of the first four adder
   //functions
   //First 2 Issue functions were registered, the
   //others not.
   RetrieveSD(radd1, iadd1, 1);
   RetrieveSD(radd2, iadd2, 1);
   RetrieveSD(radd3, iadd3, 0);
   RetrieveSD(radd4, iadd4, 0);
   RetrieveSD(radd5, iadd5, 0);
   RetrieveSD(radd6.write, iadd6.write, 0);
   RetrieveSD(radd7.write, iadd7.write, 0);
   startMulSclIssue = 1;
   while(run_Mul_Op_Issue){delay;}
   delay;
   delay;
   delay;
   delay;
   delay;
   delay;
   delay;
   delay;
   delay;
   delay;
   delay;
   delay;
   delay;
   delay;
   delay; // this cycle will be needed once read from
//radd7 is registered
   RetrieveSD(xLatR, xLatI, 0);
  }
 }

}

void HLatMulM5Wfv_Stage3(unsigned 20 offset);
```

D.3 CG_ops.hcc

```
#define MUL_LATENCY  7
#define ADD_LATENCY  6
#define SRAM_LATENCY 4

#define DOT_MUL_WAIT (SRAM_LATENCY +
#define DOT_ADD_WAIT (MUL_LATENCY + 1 + SRAM_LATENCY)

unsigned 18 halfLatticeVolume(){
 return (unsigned 18)(0 @ (NS * 12));
}

void IssueSRAMAddress(unsigned 20 address){
 par{
  zbt2_read = 1;
  zbt2_a = address;
  zbt2_c = 16;

  zbt3_read = 1;
  zbt3_a = address;
  zbt3_c = 16;

  zbt4_read = 1;
  zbt4_a = address;
  zbt4_c = 16;

  zbt5_read = 1;
  zbt5_a = address;
  zbt5_c = 16;
 }
}

void IssueSRAMFlush(){
 par{
  zbt2_c = 48;
  zbt3_c = 48;
  zbt4_c = 48;
  zbt5_c = 48;
 }
}


macro proc issueSRAMAddressesDot
 (activate, bank, address, endAddress, flush){
 par{
  if(activate){
   par{
    IssueSRAMAddress(address);

    if (address == endAddress) {
     activate = 0;
    } else {
     address++;
    }
   }
  } else {
   if(flush != 4){
    par{
     IssueSRAMFlush();
     flush++;
    }
   } else {delay;}
  }
 }
}

f_real OptimisedLatDotWfv(unsigned 20 startAddress){
 unsigned 20 dotAddress;
 unsigned 20 endAddress;
 unsigned 20 count;
 unsigned 18 operations;
 unsigned 1 runDotLoop;
 unsigned 1 issueActivate;
 unsigned 1 issueBank;
 mpram dotRAM dotResultR, dotResultI;
 unsigned 4 storeIndex;
 unsigned 32 ramIndex;
 unsigned 1 initRams, mulActivate;
 unsigned 1 addActivate, storeActivate;
 unsigned 1 addRIActivate, accActivate;
 usigned 1 accAlternate;
 unsigned 3 issueFlush;
 f_real mulOpR, mulOpI;

 //Dot product is only used with the same lattice for
 //both parameters
 f_real dotR, dotI;

 par{
  dotAddress = startAddress;
  count = 0;

  mulActivate = 0;
  addActivate = 0;
  storeActivate = 0;
  issueActivate = 1;
  addRIActivate = 0;
  accActivate = 0;
  accAlternate = 0;
  runDotLoop = 1;

  issueFlush = 0;

  ramIndex = 0;
  storeIndex = 0;

  initRams = 1;
  operations = halfLatticeVolume();
 }

 endAddress = (unsigned 20)(0 @ operations);
 endAddress += startAddress;

 do
 {
  par{
   count++;

   par{
    mulOpR = zbt2_q @ zbt3_q;
    mulOpI = zbt4_q @ zbt5_q;
   }

   //Issue Read addresses
   issueSRAMAddressesDot(issueActivate, issueBank, startAddress,
endAddress, issueFlush);

   if(count == (unsigned 20)(0 @ operations)){
    issueActivate = 0;
   } else {delay;}

   if(count == 3){
    mulActivate = 1;
   } else {delay;}

   if(count == (unsigned 20)(0 @ operations)+ 4){
    mulActivate = 0;
   } else {delay;}

   if(mulActivate){
    par{
     lmul5(mulOpR, mulOpR);
     lmul6(mulOpI, mulOpI);
    }
   } else { delay;}

   //if(count == (SRAM_LATENCY + MUL_LATENCY)){
   if(count == 11){
    addActivate = 1;
   } else {delay;}

   if(addActivate){
    par{
     if(initRams){
      par{
       la9(lm_res5, 0);
       la10(lm_res6, 0);
      }
     } else {
      par{
       //Result is ready on this cycle
       la9(lm_res5, las_res9);
       la10(lm_res6, las_res10);
      }
     }

     if(ramIndex == 6){
      par{
       initRams = 0;
       ramIndex++;
      }
     } else {
      ramIndex++;
     }
    }
   } else {delay;}

   if(count == ((unsigned 20)(0 @ operations) + 11)){
    par{
     addRIActivate = 1;
     addActivate = 0;
    }
   } else {delay;}

   if(addRIActivate){
    la9(las_res9, las_res10);
   } else {delay;}

   if(count == ((unsigned 20)(0 @ operations) + 18)){
    par{
     addRIActivate = 0;
     storeActivate = 1;
    }
   } else {delay;}

   if(count == ((unsigned 20)(0 @ operations) + 25)){
    par{
     storeActivate = 0;
     runDotLoop = 0;
    }
   } else {delay;}

   if(storeActivate){
    par{
     dotResultR.write[storeIndex] = las_res9;
     storeIndex++;
    }
   } else {delay;}
  }
 }while(runDotLoop == 1);

 la9(dotResultR.read[0], dotResultR.write[1]);
 la9(dotResultR.read[2], dotResultR.write[3]);
 la9(dotResultR.read[4], dotResultR.write[5]);
 la9(dotResultR.read[6], 0);

 delay;delay;delay;

 dotR = las_res9;
 la9(las_res9, dotR);
 dotR = las_res9;
 la9(las_res9, dotR);

 delay;delay;delay;delay;

 dotR = las_res9;
 delay;
 la9(las_res9, dotR);
 delay;delay;delay;delay;delay;delay;

 dotR = las_res9;

 return dotR;
```

```
}

macro proc IssueSclAddress(activate, address, numOps, count, alt){
 par{
  alt = !alt;

  if(activate){
   if(alt == 0){
    par{
     IssueSRAMAddress(address);
     count++;
     address++;
    }
   } else {
    par{
     if(count == numOps){
      par{
       activate = 0;
       count = 0;
      }
     } else {
      delay;
     }
    }
   }
  } else {
   if(alt == 0){
    IssueSRAMFlush();
   } else {
    delay;
   }
  }
 }
}

macro proc IssueAddAddress(activate, address, numOps, count, alt){
 par{
  alt = !alt;

  if(activate){
   if(alt == 1){
    par{
     IssueSRAMAddress(address);
     count++;
     address++;
    }
   } else {
    par{
     if(count == numOps){
      par{
       activate = 0;
       count = 0;
      }
     } else {
      delay;
     }
    }
   }
  } else {
   if(alt == 0){
    delay;
   } else {
    IssueSRAMFlush();
   }
  }
 }
}

macro proc addSclStoreIntoBRAM(activate, storeRamR, storeRamI,
ramSize, wholeLoopActivate, count){
 if(activate){
  par{
   storeRamR[count<-9] = las_res9;
   storeRamI[count<-9] = las_res10;
   count++;
  }
  par{
   if(count == ramSize){
    par{
     count = 0;
     wholeLoopActivate = 0;
    }
   } else {delay;}
  }
 } else {delay;}
}


macro proc IssueWriteAddress(address, flush){
 if(flush == 0){
  par{
   zbt2_a = address;
   zbt2_c = 0;

   zbt3_a = address;
   zbt3_c = 0;

   zbt4_a = address;
   zbt4_c = 0;

   zbt5_a = address;
   zbt5_c = 0;
  }
 } else {
  IssueSRAMFlush();
 }
}

macro proc IssueWriteData(storeRamRHH_Reg,
       storeRamRHL_Reg, storeRamRLH_Reg,
       storeRamRLL_Reg, storeRamIHH_Reg,
       storeRamIHL_Reg, storeRamILH_Reg,
       storeRamILL_Reg, doIssue){
 if(doIssue){
  par{
   zbt2_oe = 1;
   zbt2_d = storeRamRHH_Reg @ storeRamRHL_Reg;
```

```
    zbt3_oe = 1;
    zbt3_d = storeRamRLH_Reg @ storeRamRLL_Reg;

    zbt4_oe = 1;
    zbt4_d = storeRamIHH_Reg @ storeRamIHL_Reg;

    zbt5_oe = 1;
    zbt5_d = storeRamILH_Reg @ storeRamILL_Reg;
   }
  } else {delay;}
 }


macro proc storeBuffer(address, numOps, storeRamRHH,
       storeRamRHH_Reg, storeRamRHL, storeRamRHL_Reg,
       storeRamRLH, storeRamRLH_Reg, storeRamRLL,
       storeRamRLL_Reg, storeRamIHH, storeRamIHH_Reg,
       storeRamIHL, storeRamIHL_Reg, storeRamILH,
       storeRamILH_Reg, storeRamILL, storeRamILL_Reg){

 unsigned 1 doDataIssue, flushPipes, doStoreLoop;
 unsigned 1 doGetDataFromRams, stage2;
 unsigned 11 count;
 unsigned 10 index;

 par{
  doDataIssue = 0;
  flushPipes = 0;
  doStoreLoop = 1;
  index = 0;
  count = 1;
  stage2 = 0;
 }

 par{
  do{
   par{
    IssueWriteAddress(address, flushPipes);
    count++;
   }
  } while(doStoreLoop);

  do{
   par{
    if(doGetDataFromRams){
     par{
      index++;
      storeRamRHH_Reg = storeRamRHH[index];
      storeRamRHL_Reg = storeRamRHL[index];
      storeRamRLH_Reg = storeRamRLH[index];
      storeRamRLL_Reg = storeRamRLL[index];

      storeRamIHH_Reg = storeRamIHH[index];
      storeRamIHL_Reg = storeRamIHL[index];
      storeRamILH_Reg = storeRamILH[index];
      storeRamILL_Reg = storeRamILL[index];
     }
    } else {
     delay;
    }
   }
  } while(doStoreLoop);

  do{
   par{
    IssueWriteData(storeRamRHH_Reg, storeRamRHL_Reg,
storeRamRLH_Reg, storeRamRLL_Reg,
        storeRamIHH_Reg, storeRamIHL_Reg, storeRamILH_Reg,
storeRamILL_Reg,
        doDataIssue);
   }
  }while(doStoreLoop);

  do{
   par{
    doGetDataFromRams = !flushPipes;
    doDataIssue = doGetDataFromRams;

    if((doDataIssue == 0) && (count[10] == 1)){
     doStoreLoop = 0;
    } else {delay;}

    //if(count == 1024) --- This can be done by testing bit 9 of
count instead
    if(count == numOps){
     flushPipes = 1;
    } else {
     delay;
    }

    if(!flushPipes){
     address++;
    } else {delay;}
   }
  } while(doStoreLoop);
 }
}

unsigned 11 GetIterationCount
    (unsigned 12 mainCount, unsigned 18 operations,
    unsigned 11 blockRamSize){

 if((mainCount == 1) && (operations<-10 != 0)){
  return (unsigned 11)(0 @ operations<-10);
 } else {
  return blockRamSize;
 }
}

void OptimisedLatAddSclWfvWfv
       (unsigned 20 startOpAddress,
        unsigned 20 startResAddress,
        unsigned 1 functionType,
        f_real scl){

 unsigned 20 sclAddress, addAddress, storeAddress;
 unsigned 1 runMainLoop, activeSclAddress, activeAddAddress;
 unsigned 1 runAddSclLoop, runStoreLoop;
 unsigned 1 ramOutActivate;
 unsigned 1 issueSclAlt, issueAddAlt;
 f_real banks2n3, banks4n5;
 f_real banks2n3Piped, banks4n5Piped;
 f_real las_res10_reg, las_res9_reg;

 ram unsigned 16 storeRamRHH[1024], storeRamRHL[1024]
  with {block = "BlockRAM"};
 ram unsigned 16 storeRamRLH[1024], storeRamRLL[1024]
  with {block = "BlockRAM"};
 ram unsigned 16 storeRamIHH[1024], storeRamIHL[1024]
  with {block = "BlockRAM"};
 ram unsigned 16 storeRamILH[1024], storeRamILL[1024]
  with {block = "BlockRAM"};
 unsigned 16 storeRamRHH_Reg, storeRamRHL_Reg;
 unsigned 16 storeRamRLH_Reg, storeRamRLL_Reg;
 unsigned 16 storeRamIHH_Reg, storeRamIHL_Reg;
 unsigned 16  storeRamILH_Reg, storeRamILL_Reg;

 unsigned 11 countSclIssue, countAddIssue;
 unsigned 11 countRamStore;

 unsigned 18 operations;

 unsigned 12 count, mainCount;

 const unsigned 11 blockRamSize = 1024;

 unsigned 11 iterationCount;

 par{
  if(functionType == 0){ //LatAddSclWfvWfv
   par{
    sclAddress = startOpAddress;
    addAddress = startResAddress;
   }
  } else {    //LatSclWfvAddWfv
   par{
    sclAddress = startResAddress;
    addAddress = startOpAddress;
   }
  }

  storeAddress = startResAddress;
  operations = halfLatticeVolume();

  count = 0;

  countSclIssue = 0;
  countAddIssue = 0;
  countRamStore = 0;

  runMainLoop = 1;
  runAddSclLoop = 1;
  activeSclAddress = 1;
  activeAddAddress = 0;
  ramOutActivate = 0;
  issueSclAlt = 0;
  issueAddAlt = 0;

  runStoreLoop = 0;
 }

 mainCount = (unsigned 12)(0 @ operations\\10) -
  (unsigned 12)(0 @ (operations<-10 == 0));

 iterationCount = GetIterationCount
           (mainCount, operations, blockRamSize);

 do{
  par{
   do{
    par{
     count++;

     if(count == 6){
      activeAddAddress = 1;
     } else {delay;}

     if(count == 20){
      ramOutActivate = 1;
     } else {delay;}
    }
   } while(runAddSclLoop);

   do{
    IssueSclAddress
     (activeSclAddress, sclAddress, iterationCount,
     countSclIssue, issueSclAlt);
   }while(runAddSclLoop);

   do{
    IssueAddAddress
     (activeAddAddress, addAddress, iterationCount,
     countAddIssue, issueAddAlt);
   }while(runAddSclLoop);

   do{
    par{
     ReadVar2 = zbt2_q;
     ReadVar3 = zbt3_q;
     ReadVar4 = zbt4_q;
     ReadVar5 = zbt5_q;
    }
   } while(runAddSclLoop);

   do{
    par{
     banks2n3 = ReadVar2 @ ReadVar3;
     banks4n5 = ReadVar4 @ ReadVar5;
    }
   } while (runAddSclLoop);

   do{
    par{
```

```
  banks4n5Piped = banks4n5;
  banks2n3Piped = banks2n3;
 }
} while(runAddSclLoop);

do{
 par{
  lmul3(scl, banks2n3);
  lmul4(scl, banks4n5);
 }
} while (runAddSclLoop);

do{
 par{
  la9(lm_res3, banks2n3Piped);
  la10(lm_res4, banks4n5Piped);
 }
} while(runAddSclLoop);


do{
 par{
  las_res9_reg = las_res9;
  las_res10_reg = las_res10;
 }
} while(runAddSclLoop);

do{
 if(ramOutActivate){
  par{
   storeRamRHH[countRamStore<-10] =
   las_res9_reg[63:48];
   storeRamRHL[countRamStore<-10] =
   las_res9_reg[47:32];
   storeRamRLH[countRamStore<-10] =
   las_res9_reg[31:16];
   storeRamRLL[countRamStore<-10] =
   las_res9_reg[15:0];

   storeRamIHH[countRamStore<-10] =
   las_res10_reg[63:48];
   storeRamIHL[countRamStore<-10] =
   las_res10_reg[47:32];
   storeRamILH[countRamStore<-10] =
   las_res10_reg[31:16];
   storeRamILL[countRamStore<-10] =
   las_res10_reg[15:0];

   countRamStore++;
  }

  par{
   if(countRamStore == iterationCount){
    par{
     countRamStore = 0;
     runAddSclLoop = 0;
    }
   } else {delay;}
  }
```

```
    } else {delay;}

   } while(runAddSclLoop);
  }

  storeBuffer(storeAddress, iterationCount,
    storeRamRHH, storeRamRHH_Reg, storeRamRHL,
    storeRamRHL_Reg, storeRamRLH, storeRamRLH_Reg,
    storeRamRLL, storeRamRLL_Reg, storeRamIHH,
    storeRamIHH_Reg, storeRamIHL, storeRamIHL_Reg,
    storeRamILH, storeRamILH_Reg, storeRamILL,
    storeRamILL_Reg);

  if(mainCount == 0){
   runMainLoop = 0;
  } else {
   par{
    iterationCount = GetIterationCount
      (mainCount, operations, blockRamSize);
    mainCount--;
    count = 0;
    countSclIssue = 0;
    countAddIssue = 0;
    countRamStore = 0;

    runMainLoop = 1;
    runAddSclLoop = 1;
    activeSclAddress = 1;
    activeAddAddress = 0;
    ramOutActivate = 0;

    issueSclAlt = 0;
    issueAddAlt = 0;

    runStoreLoop = 0;
   }
  }

 } while(runMainLoop);
```

D.4 SRAM_functions.hcc

```
//qcdlog32
//Variables for SRAM macro procs
unsigned 32 ReadVar0, ReadVar1, ReadVar2, ReadVar3, ReadVar4,
ReadVar5;
unsigned 32 WriteVar0, WriteVar1, WriteVar2, WriteVar3, WriteVar4,
WriteVar5;

//Contents of Srams.hcc
unsigned 32 ReadBank0(unsigned 20 address){
 par{
  zbt0_read = 1;
  zbt0_a = address;
  zbt0_c = 16;
 }

 par{
  zbt0_c = 48;
 }

 par{
  zbt0_c = 48;
 }

 par{
  zbt0_c = 48;
  ReadVar0 = zbt0_q;
 }
 return ReadVar0;
}

void WriteBank0(unsigned 20 address, unsigned 32 out){

 par{
  zbt0_a = address;
  zbt0_c = 0;
 }

 par{
  zbt0_c = 48;
 }

 par{
  zbt0_c = 48;
  zbt0_oe = 1;
  zbt0_d = (unsigned 32)(0 @ out);
 }

 //Insert delay to prevent an attempted read
 // before SRAM has read and updated value
 par{
  delay;
 }
}

unsigned 32 ReadBank1(unsigned 20 address){
 par{
  zbt1_read = 1;
  zbt1_a = address;
  zbt1_c = 16;
 }

 par{
  zbt1_c = 48;
 }

 par{
  zbt1_c = 48;
 }

 par{
  zbt1_c = 48;
  ReadVar1 = zbt1_q;
 }
 return ReadVar1;
}

void WriteBank1(unsigned 20 address, unsigned 32 out){
 par{
  zbt1_a = address;
  zbt1_c = 0;
 }

 par{
  zbt1_c = 48;
 }

 par{
  zbt1_c = 48;
  zbt1_oe = 1;
  zbt1_d = (unsigned 32)(0 @ out);
 }

 //Insert delay to prevent an attempted read
 // before SRAM has read and updated value
 par{
  delay;
 }
}


unsigned 32 ReadBank2(unsigned 20 address){
 par{
  zbt2_read = 1;
  zbt2_a = address;
  zbt2_c = 16;
 }

 par{
  zbt2_c = 48;
 }

 par{
  zbt2_c = 48;
 }
 par{
  zbt2_c = 48;
  ReadVar2 = zbt2_q;
 }
 return ReadVar2;
}

void WriteBank2(unsigned 20 address, unsigned 32 out){
 par{
  zbt2_a = address;
  zbt2_c = 0;
 }

 par{
  zbt2_c = 48;
 }

 par{
  zbt2_c = 48;
  zbt2_oe = 1;
  zbt2_d = (unsigned 32)(0 @ out);
 }

 //Insert delay to prevent an attempted read
 // before SRAM has read and updated value
 par{
  delay;
 }
}

unsigned 32 ReadBank3(unsigned 20 address){
 par{
  zbt3_read = 1;
  zbt3_a = address;
  zbt3_c = 16;
 }

 par{
  zbt3_c = 48;
 }

 par{
  zbt3_c = 48;
 }

 par{
  zbt3_c = 48;
  ReadVar3 = zbt3_q;
 }
 return ReadVar3;
}

void WriteBank3(unsigned 20 address, unsigned 32 out){
 par{
  zbt3_a = address;
  zbt3_c = 0;
 }
 par{
  zbt3_c = 48;
 }

 par{
  zbt3_c = 48;
  zbt3_oe = 1;
  zbt3_d = (unsigned 32)(0 @ out);
 }

 //Insert delay to prevent an attempted read
 // before SRAM has read and updated value
 par{
  delay;
 }
}


unsigned 32 ReadBank4(unsigned 20 address){
 par{
  zbt4_read = 1;
  zbt4_a = address;
  zbt4_c = 16;
 }

 par{
  zbt4_c = 48;
 }

 par{
  zbt4_c = 48;
 }

 par{
  zbt4_c = 48;
  ReadVar4 = zbt4_q;
 }
 return ReadVar4;
}

void WriteBank4(unsigned 20 address, unsigned 32 out){
 par{
  zbt4_a = address;
  zbt4_c = 0;
 }

 par{
  zbt4_c = 48;
 }

 par{
  zbt4_c = 48;
  zbt4_oe = 1;
  zbt4_d = (unsigned 32)(0 @ out);
 }

 //Insert delay to prevent an attempted read
 // before SRAM has read and updated value
 par{
  delay;
 }
}


unsigned 32 ReadBank5(unsigned 20 address){
 par{
  zbt5_read = 1;
  zbt5_a = address;
  zbt5_c = 16;
 }

 par{
  zbt5_c = 48;
 }

 par{
  zbt5_c = 48;
 }

 par{
  zbt5_c = 48;
  ReadVar5 = zbt5_q;
 }
 return ReadVar5;
}

void WriteBank5(unsigned 20 address, unsigned 32 out){
 par{
  zbt5_a = address;
  zbt5_c = 0;
 }

 par{
  zbt5_c = 48;
 }

 par{
  zbt5_c = 48;
  zbt5_oe = 1;
  zbt5_d = (unsigned 32)(0 @ out);
 }

 //Insert delay to prevent an attempted read
 // before SRAM has read and updated value
 par{
  delay;
 }
}

unsigned 32 ReadWfv0, ReadWfv1, ReadWfv2, ReadWfv3, ReadWfv4,
ReadWfv5;

unsigned 1 ReadWfvLoop;
unsigned 1 ReadWfvAlt;
unsigned 4 ReadWfvA;
unsigned 20 ReadWfvAddress;
unsigned 1 ReadGl3Loop;
unsigned 1 ReadGl3Alt;
unsigned 5 ReadGl3A;
unsigned 20 ReadGl3Address;


void ReadGl3(unsigned 20 offset, unsigned 20 s, unsigned 2 t, f_real
*rout, unsigned 4 *rIndex, f_real *rReg,
                 f_real *iout, unsigned 4 *iIndex, f_real *iReg){
 //unsigned 20 rIndex, iIndex;
 //f_real rReg, iReg;

 par{
  ReadGl3Address = (unsigned 20)(0 @ offset) + ((unsigned 20)(0 @ s)
* 72) + ((unsigned 20)(0 @ t) * 18);
  ReadGl3A = 0;
  *iIndex = 0;
  *rIndex = 0;
  ReadGl3Loop = 1;
 }

 while(ReadGl3Loop){
  par{
   ReadWfv0 = ReadBank0(ReadGl3Address);
   ReadWfv1 = ReadBank1(ReadGl3Address);
  }

  par{
   *rReg = ReadWfv0 @ ReadWfv1;
   ReadGl3Address++;
  }

  par{
   ReadWfv0 = ReadBank0(ReadGl3Address);
   ReadWfv1 = ReadBank1(ReadGl3Address);
  }

  par{
   *iReg = ReadWfv0 @ ReadWfv1;
   ReadGl3Address++;
  }

  par{
   if(ReadGl3A == 8){
    ReadGl3Loop = 0;
   } else {
    par{
     ReadGl3A++;
    }
   }
   rout[*rIndex] = *rReg;
   iout[*iIndex] = *iReg;
   *rIndex = *rIndex+1;
   *iIndex = *iIndex+1;
  }
 }
}

unsigned 20 WriteGl3Address;
unsigned 5 WriteGl3Count;
unsigned 1 WriteGl3Loop;
unsigned 1 WriteGl3Alt;

//s is the site number of the wfv being written
//multiply by 4 to get the bank 0 address for it
void WriteGl3(unsigned 20 offset, unsigned 20 s,
    unsigned 2 t, f_real *rout, unsigned 4 *rIndex,
    f_real *rReg, f_real *iout, unsigned 4 *iIndex,
    f_real *iReg){

 par{
  WriteGl3Address = (unsigned 20)(0 @ offset) +
   ((unsigned 20)(0 @ s) * 72) +
   ((unsigned 20)(0 @ t) * 18);
  WriteGl3Count = 0;
  WriteGl3Loop = 1;
  *rIndex = 0;
  *iIndex = 0;
 }

 while(WriteGl3Loop){
  par{
   *rReg = rout[*rIndex];
   *iReg = iout[*iIndex];
   *rIndex = *rIndex+1;
   *iIndex = *iIndex+1;
  }

  par{
   ReadWfv0 = *rReg\\32;
   ReadWfv1 = *rReg<-32;
  }
  par{
   WriteBank0(WriteGl3Address, ReadWfv0);
   WriteBank1(WriteGl3Address, ReadWfv1);
  }

  par{
   WriteGl3Address++;
   ReadWfv0 = *iReg\\32;
   ReadWfv1 = *iReg<-32;
  }
  par{
   WriteBank0(WriteGl3Address, ReadWfv0);
   WriteBank1(WriteGl3Address, ReadWfv1);
  }

  if(WriteGl3Count == 8){
   WriteGl3Loop = 0;
  } else {
   par{
    WriteGl3Address++;
    WriteGl3Count++;
   }
  }
 }
}
```

```
unsigned 5 RGcycles;
unsigned 5 RGa;
unsigned 20 RGaddress;
unsigned 1 RGaloop, RGbloop, RGIssue;
unsigned 1 RGFlush, RGRet, RGArrDone;
unsigned 17 interGl3;

void calcGl3AddressStage1(unsigned 16 s){
 interGl3 = (unsigned 17)(0 @ s) * 9;
}

void calcGl3AddressStage2(unsigned 20 t){
 const unsigned 3 threeBitZero = 0;

 RGaddress = ((interGl3 @ threeBitZero) + t);
}

void RGl3setup(unsigned 16 s, unsigned 20 t){
 par{
  calcGl3AddressStage1(s);
  RGa = 0;
  RGaloop = 1;
  RGbloop = 0;
  RGArrDone = 0;
 }
 calcGl3AddressStage2(t);
}

void RGl3flush(){
 par{
  zbt0_c = 48;
  zbt1_c = 48;
 }
 par{
  zbt0_c = 48;
  zbt1_c = 48;
 }
 par{
  zbt0_c = 48;
  zbt1_c = 48;
 }
}

void IssueRGl3(unsigned 16 snext,
    unsigned 20 tnext, unsigned 1 odd){
 unsigned 1 RGodd;

 do{
  par{
   zbt0_read = 1;
   zbt0_a = RGaddress;
   zbt0_c = 16;

   zbt1_read = 1;
   zbt1_a = RGaddress;
   zbt1_c = 16;

   if(RGa == 16){
    par{
     RGArrDone = 1;
     calcGl3AddressStage1(snext);
    }
   } else {
    delay;
   }

   if(RGArrDone){
    par{
     RGa = 0;
     RGaloop = !RGaloop;
     RGArrDone = 0;
     calcGl3AddressStage2(tnext);
    }
   } else {
    par{
     RGodd = odd;
     RGaddress++;
     RGa++;
    }
   }
  }
 }while (RGaloop == RGodd);
}

unsigned 5 retRGa;
unsigned 5 retRGaInit;
unsigned 5 retRGCount;
unsigned 1 retRGaloop, retRGbloop;
unsigned 1 retGl3Alt;

void setupRetRGl3(unsigned 1 highSpace){
 par{
  if(highSpace == 0){
   par{
    retRGa = 0;
    retRGaInit = 0;
   }
  } else {
   par{
    retRGa = 9;
    retRGaInit = 9;
   }
  }
  retRGCount = 0;
  retRGaloop = 1;
  retRGbloop = 0;
  retGl3Alt = 0;
 }
}

void RetRGl3(f_real *rout, f_real *iout,
  unsigned 1 odd){
 unsigned 1 retRGodd;
 do{
  par{
   if(retGl3Alt == 0){
    rout[retRGa] = ReadVar0 @ ReadVar1;
   } else {
    iout[retRGa] = ReadVar0 @ ReadVar1;
   }

   retGl3Alt = !retGl3Alt;

   if(retRGCount == 17){
    par{
     retRGa = retRGaInit;
     retRGaloop = !retRGaloop;
     retRGCount = 0;
    }
   } else {
    par{
     if(retGl3Alt == 1){
      retRGa++;
     } else {delay;}
     retRGCount++;
     retRGodd = odd;
    }
   }
  }
 }while (retRGaloop == retRGodd);
}


void ReadWfv(unsigned 20 offset, unsigned 20 s, f_real
  *rout, unsigned 4 *rIndex, f_real *rReg, f_real
  *iout, unsigned 4 *iIndex, f_real *iReg){

 par{
  ReadWfvAddress = (unsigned 20)(0 @ offset) +
   ((unsigned 20)(0 @ (s<-18 * 12)));
  ReadWfvA = 0;
  *iIndex = 0;
  *rIndex = 0;
  ReadWfvLoop = 1;
 }

 while(ReadWfvLoop){
  par{
   ReadWfv2 = ReadBank2(ReadWfvAddress);
   ReadWfv3 = ReadBank3(ReadWfvAddress);
   ReadWfv4 = ReadBank4(ReadWfvAddress);
   ReadWfv5 = ReadBank5(ReadWfvAddress);
  }

  par{
   *rReg = ReadWfv2 @ ReadWfv3;
   *iReg = ReadWfv4 @ ReadWfv5;
  }

  par{
   rout[*rIndex] = *rReg;
   iout[*iIndex] = *iReg;
   *rIndex = *rIndex+1;
   *iIndex = *iIndex+1;
  }

  par{
   if(ReadWfvA == 11){
    ReadWfvLoop = 0;
   } else {
    par{
     ReadWfvA++;
     ReadWfvAddress++;
    }
   }
  }
 }
}



unsigned 20 WriteWfvAddress;
unsigned 4 WriteWfvCount;
unsigned 1 WriteWfvLoop;

void WriteWfv(unsigned 20 offset, unsigned 20 s,
   f_real *rout, unsigned 4 *rIndex, f_real *rReg,
   f_real *iout, unsigned 4 *iIndex, f_real *iReg){

 par{
  WriteWfvAddress = (unsigned 20)(0 @ offset) +
   ((unsigned 20)(0 @ (s<-18 * 12)));
  WriteWfvCount = 0;
  WriteWfvLoop = 1;
  *rIndex = 0;
  *iIndex = 0;
 }

 while(WriteWfvLoop){
  par{
   *rReg = rout[*rIndex];
   *iReg = iout[*iIndex];
   *rIndex = *rIndex+1;
   *iIndex = *iIndex+1;
  }

  par{
   ReadWfv2 = *rReg\\32;
   ReadWfv3 = *rReg<-32;
   ReadWfv4 = *iReg\\32;
   ReadWfv5 = *iReg<-32;
  }

  par{
   WriteBank2(WriteWfvAddress, ReadWfv2);
   WriteBank3(WriteWfvAddress, ReadWfv3);
   WriteBank4(WriteWfvAddress, ReadWfv4);
   WriteBank5(WriteWfvAddress, ReadWfv5);
  }

  par{
   if(WriteWfvCount == 11){
    WriteWfvLoop = 0;
   }
   WriteWfvAddress++;
   WriteWfvCount++;
   }
  }
 }

unsigned 20 RWaddress;
unsigned 1 RWaloop, RWbloop, RWIssue, RWFlush;
unsigned 1 RWRet, RWArrDone;
unsigned 4 RWa;
unsigned 17 interWfv;

void calcWfvAddressStage1(unsigned address){
 interWfv = (unsigned 17)(0 @ address) * 12;
}

unsigned 20 calcWfvAddressStage2(unsigned 20 offset){
 return (unsigned 20)(0 @ interWfv) + offset;
}
void RWfvsetup(unsigned 16 address, unsigned 20 offset){
 par{
  calcWfvAddressStage1(address);
  RWa = 0;
  RWaloop = 1;
  RWbloop = 0;
  RWArrDone = 0;
 }
 RWaddress = calcWfvAddressStage2(offset);
}

void flushWfvPipes(){
 par{
  zbt2_c = 48;
  zbt3_c = 48;
  zbt4_c = 48;
  zbt5_c = 48;
 }
}

void RWfvflush(){
 flushWfvPipes();
 flushWfvPipes();
 flushWfvPipes();
}

void IssueRWfv(unsigned 16 anext, unsigned 20 offset,
  unsigned 1 odd){
 unsigned 1 RWodd;

 do{
  par{
   par{
    zbt2_read = 1;
    zbt2_a = RWaddress;
    zbt2_c = 16;

    zbt3_read = 1;
    zbt3_a = RWaddress;
    zbt3_c = 16;

    zbt4_read = 1;
    zbt4_a = RWaddress;
    zbt4_c = 16;

    zbt5_read = 1;
    zbt5_a = RWaddress;
    zbt5_c = 16;
   }

   if(RWa == 10){
    par{
     RWArrDone = 1;
     calcWfvAddressStage1(anext);
    }
   } else {delay;}

   if(RWArrDone){
    par{
     RWa = 0;
     RWaloop = !RWaloop;
     RWArrDone = 0;
     RWaddress = calcWfvAddressStage2(offset);
    }
   } else {
    par{
     RWaddress++;
     RWa++;
     RWodd = odd;
    }
   }
  }
 }while (RWaloop == RWodd);
}

unsigned 4 retRWa;
unsigned 1 retRWaloop, retRWbloop;

void setupRetRWfv(){
 par{
  retRWa = 0;
  retRWaloop = 1;
  retRWbloop = 0;
 }
}

void RetRWfv(f_real *rout, f_real *iout,
              unsigned 1 odd){
 unsigned 1 retRWodd;

 do{
  par{
   rout[retRWa] = ReadVar2 @ ReadVar3;
   iout[retRWa] = ReadVar4 @ ReadVar5;

   if(retRWa == 11){
    par{
     retRWa = 0;
     retRWaloop = !retRWaloop;
    }
   } else {
    par{
     retRWa++;
     retRWodd = odd;
    }
   }
  }
 }while (retRWaloop == retRWodd);
}

unsigned XWIDTH zS2, yS2, yS3, yS4, xS2, xS3, xS4, xS5, xS6;
unsigned 8 res1, res2;
unsigned 12 res3, res4;
unsigned 16 res5, res6;

shared expr siteCalc1(x, y, z, t) =
  ((unsigned 8)(0 @ t) * (unsigned 8)(0 @ NZ));
shared expr siteCalc2() =
  res1 + (unsigned 8)(0 @ zS2);

shared expr siteCalc3() =
  ((unsigned 12)(0 @ res2) * (unsigned 12)(0 @ NY));
shared expr siteCalc4() =
  res3 + (unsigned 12)(0 @ yS4);

shared expr siteCalc5() =
  ((unsigned 16)(0 @ res4) * (unsigned 16)(0 @ NX));
shared expr siteCalc6() =
  res5 + (unsigned 16)(0 @ xS6);

/*Originals used the mod operator (very slow) could be replaced
using if statements
 much faster
*/
//shared expr p1(xIn, NXIn) = (xIn+1) % NXIn;

unsigned 4 p1(unsigned 4 xIn, unsigned 4 NXIn){
 if((xIn + 1) == NXIn) {
  return 0;
 } else {
  return xIn + 1;
 }
}


//shared expr m1(xIn, NXIn) = ((xIn+NXIn)-1) % NXIn;

unsigned 4 m1(unsigned 4 xIn, unsigned 4 NXIn){
 if(xIn == 0) {
  return NXIn -1;
 } else {
  return xIn -1;
 }
}

void siteStage1(unsigned XWIDTH x, unsigned XWIDTH y, unsigned
XWIDTH z, unsigned XWIDTH t){
 par{
  zS2 = z;
  yS2 = y;
  xS2 = x;
  res1 = siteCalc1(x, y, z, t);
 }
}

void siteStage2(){
 par{
  yS3 = yS2;
  xS3 = xS2;
  res2 = siteCalc2();
 }
}

void siteStage3(){
 par{
  yS4 = yS3;
  xS4 = xS3;
  res3 = siteCalc3();
 }
}

void siteStage4(){
 par{
  xS5 = xS4;
  res4 = siteCalc4();
 }
}

void siteStage5(){
 par{
  xS6 = xS5;
  res5 = siteCalc5();
 }
}

void siteStage6(){
 par{
  res6 = siteCalc6();
 }
}


void sitePiped(unsigned XWIDTH x, unsigned XWIDTH y,
               unsigned XWIDTH z, unsigned XWIDTH t){
 par{
  siteStage1(x, y, z, t);
  siteStage2();
  siteStage3();
  siteStage4();
  siteStage5();
  siteStage6();
 }
}

shared expr CastRes6() = (unsigned 16)(0 @ res6);

void CalculateOffsetsPiped(){
  par{
   xp1 = p1(x, NX);
   xm1 = m1(x, NX);
   sitePiped(x, y, z, t);
  }
  par{
   yp1 = p1(y, NY);
   ym1 = m1(y, NY);
   sitePiped(xm1, y, z, t);
  }
  par{
   zp1 = p1(z, NZ);
   zm1 = m1(z, NZ);
   sitePiped(x, ym1, z, t);
  }
  par{
   tp1 = p1(t, NT);
   tm1 = m1(t, NT);
   sitePiped(x, y, zm1, t);
  }

  par{
   seq{
    sitePiped(x, y, z, tm1);
    sitePiped(xp1, y, z, t);
    sitePiped(x, yp1, z, t);
    sitePiped(x, y, zp1, t);
    sitePiped(x, y, z, tp1);
    par{
     siteStage2();
     siteStage3();
     siteStage4();
     siteStage5();
     siteStage6();
    }
    par{
     siteStage3();
     siteStage4();
     siteStage5();
     siteStage6();
    }
    par{
     siteStage4();
     siteStage5();
     siteStage6();
    }
    par{
     siteStage5();
     siteStage6();
    }
    siteStage6();
   }

   seq{
    delay;
    delay;
    rSite = CastRes6();
    smx = CastRes6();
    smy = CastRes6();
    smz = CastRes6();
    smt = CastRes6();
```

```
    spx = CastRes6();
    spy = CastRes6();
    spz = CastRes6();
    spt = CastRes6();
   }
  }
  delay;
}

void UpdateLoopCounters(){
 numNS++;
 if(x==(NX-1)){
  x=0;
  if(y==(NY-1)) {
   y=0;
   if(z==(NZ-1)){
    z=0;
    if(t==(NT-1)){
     t=0;
     iter++;
    } else {
     t++;
    }
   } else {
    z++;
   }
  } else {
   y++;
  }
 } else {
  x++;
 }

}

unsigned 1 wfvAllDone;

void ReadWfvAll(unsigned 20 offset_P){
 unsigned 20 offset;

 par{
  wfvAllDone=0;
  offset = offset_P;
 }

 par{
  do{
   par{
    ReadVar2 = zbt2_q;
    ReadVar3 = zbt3_q;
    ReadVar4 = zbt4_q;
    ReadVar5 = zbt5_q;
   }
  }while(wfvAllDone == 0);
  //Issue addresses to RAMS
  seq{
   RWfvsetup(smx, offset);
   IssueRWfv(smy, offset, 1);
   IssueRWfv(smz, offset, 0);
   IssueRWfv(smt, offset, 1);
   IssueRWfv(spx, offset, 0);
   IssueRWfv(spy, offset, 1);
   IssueRWfv(spz, offset, 0);
   IssueRWfv(spt, offset, 1);
   IssueRWfv(rSite, offset, 0);
   IssueRWfv(smx, offset, 1);
   RWfvflush();
  }
  seq{
   setupRetRWfv();
   delay; //To compensate for the 2 cycle addres calculation
   delay;delay;delay;delay;
   RetRWfv(yLat1R.write, yLat1I.write, 1);
   RetRWfv(yLat3R, yLat3I, 0);
   RetRWfv(yLat5R.write, yLat5I.write, 1);
   RetRWfv(yLat7R, yLat7I, 0);
   RetRWfv(yLat0R.write, yLat0I.write, 1);
   RetRWfv(yLat2R, yLat2I, 0);
   RetRWfv(yLat4R.write, yLat4I.write, 1);
   RetRWfv(yLat6R, yLat6I, 0);
   RetRWfv(yLat8R, yLat8I, 1);
   wfvAllDone = 1;
  }
 }
}

unsigned 4 gLat0R_Index, gLat0I_Index;
unsigned 4 gLat1R_Index, gLat1I_Index;

//These constants are the amount to be addres for none,
//one two and three gl3 matrices
const unsigned 20 oneC = 18, two = 36;
const unsigned 20 three = 54, zero = 0;

unsigned 1 gl3alldone;

void ReadGl3All(){
 gl3alldone =0;
 par{
  do{
   par{
    ReadVar0 = zbt0_q;
    ReadVar1 = zbt1_q;
   }
  }while(gl3alldone == 0);
  //Issue addresses to RAMS
  seq{
   RGl3setup(smx, zero);
   IssueRGl3(smy, oneC, 1);
   IssueRGl3(smz, two, 0);
   IssueRGl3(smt, three, 1);
   IssueRGl3(rSite, zero, 0);
   IssueRGl3(rSite, oneC, 1);
   IssueRGl3(rSite, two, 0);
   IssueRGl3(rSite, three, 1);
   IssueRGl3(smz, two, 0);
   RGl3flush();
  }
  seq{
   setupRetRGl3(ReadGl3AltRamHalf);
   delay; //Compensate for 2 cycle addres calculation
   delay;delay;delay;delay;
   RetRGl3(gLat1R.write, gLat1I.write, 1);
   RetRGl3(gLat3R.write, gLat3I.write, 0);
   RetRGl3(gLat5R.write, gLat5I.write, 1);
   RetRGl3(gLat7R.write, gLat7I.write, 0);
   RetRGl3(gLat0R.write, gLat0I.write, 1);
   RetRGl3(gLat2R.write, gLat2I.write, 0);
   RetRGl3(gLat4R.write, gLat4I.write, 1);
   RetRGl3(gLat6R.write, gLat6I.write, 0);
   par{
    gl3alldone = 1;
    ReadGl3AltRamHalf = !ReadGl3AltRamHalf;
    ReadGl3AltRamHalf_Stage1 = ReadGl3AltRamHalf;
   }
  }
 }
}

macro proc ReadOperands(offset){
 ReadWfv(offset, (unsigned 20)(0 @ spx), yLat0R.write,
  &yLatR_Index, &yLat0R_Write_In, yLat0I.write,
  &yLatI_Index, &yLat0I_Write_In);
 ReadWfv(offset, (unsigned 20)(0 @ smx), yLat1R.write,
  &yLatR_Index, &yLat1R_Write_In, yLat1I.write,
  &yLatI_Index, &yLat1I_Write_In);
 ReadWfv(offset, (unsigned 20)(0 @ spy), yLat2R,
  &yLatR_Index, &yLat2R_Read_In, yLat2I, &yLatI_Index,
  &yLat2I_Read_In);
 ReadWfv(offset, (unsigned 20)(0 @ smy), yLat3R,
  &yLatR_Index, &yLat3R_Read_In, yLat3I, &yLatI_Index,
  &yLat3I_Read_In);
 ReadWfv(offset, (unsigned 20)(0 @ spz), yLat4R.write,
  &yLatR_Index, &yLat4R_Write_In, yLat4I.write,
  &yLatI_Index, &yLat4I_Write_In);
 ReadWfv(offset, (unsigned 20)(0 @ smz), yLat5R.write,
  &yLatR_Index, &yLat5R_Write_In, yLat5I.write,
  &yLatI_Index, &yLat5I_Write_In);
 ReadWfv(offset, (unsigned 20)(0 @ spt), yLat6R,
  &yLatR_Index, &yLat6R_Read_In, yLat6I, &yLatI_Index,
  &yLat6I_Read_In);
 ReadWfv(offset, (unsigned 20)(0 @ smt), yLat7R,
  &yLatR_Index, &yLat7R_Read_In, yLat7I, &yLatI_Index,
  &yLat7I_Read_In);
 ReadWfv(offset, (unsigned 20)(0 @ rSite), yLat8R,
  &yLatR_Index, &yLat8R_In, yLat8I, &yLatI_Index,
  &yLat8I_In);

 ReadGl3(G_OFFSET, (unsigned 20)(0 @ rSite), G_ZERO,
  gLat0R.write, &gLat0R_Index, &gLat0R_In,
  gLat0I.write, &gLat0I_Index, &gLat0I_In);
 ReadGl3(G_OFFSET, (unsigned 20)(0 @ smx), G_ZERO,
  gLat1R.write, &gLat1R_Index, &gLat1R_In,
  gLat1I.write, &gLat1I_Index, &gLat1I_In);
 ReadGl3(G_OFFSET, (unsigned 20)(0 @ rSite), G_ONE,
  gLat2R.write, &gLat0R_Index, &gLat2R_In,
  gLat2I.write, &gLat0I_Index, &gLat2I_In);
 ReadGl3(G_OFFSET, (unsigned 20)(0 @ smy), G_ONE,
  gLat3R.write, &gLat1R_Index, &gLat3R_In,
  gLat3I.write, &gLat1I_Index, &gLat3I_In);
 ReadGl3(G_OFFSET, (unsigned 20)(0 @ rSite), G_TWO,
  gLat4R.write, &gLat0R_Index, &gLat4R_In,
  gLat4I.write, &gLat0I_Index, &gLat4I_In);
 ReadGl3(G_OFFSET, (unsigned 20)(0 @ smz), G_TWO,
  gLat5R.write, &gLat1R_Index, &gLat5R_In,
  gLat5I.write, &gLat1I_Index, &gLat5I_In);
 ReadGl3(G_OFFSET, (unsigned 20)(0 @ rSite), G_THREE,
  gLat6R.write, &gLat0R_Index, &gLat6R_In,
  gLat6I.write, &gLat0I_Index, &gLat6I_In);
 ReadGl3(G_OFFSET, (unsigned 20)(0 @ smt), G_THREE,
  gLat7R.write, &gLat1R_Index, &gLat7R_In,
  gLat7I.write, &gLat1I_Index, &gLat7I_In);
}

unsigned 6 ROPcount;
unsigned 1 runMainLoop;

//use to signal to ReadOperands that
//resultwrite has completed

unsigned 1 stage3_running;
unsigned 20 ROPoffset;
void ReadOperandsPiped(unsigned 20 offset){

 //must wait 12 cycles for
 //gamma functions to use all y operands
par{
  ROPcount = 0;
  ROPoffset = offset;
 }

 CalculateOffsetsPiped();

 do{
  delay;
 }while(stage3_running);

 //Calculate offsets takes 12 cycles to complete

 par{
  seq{
   ReadWfvAll(offset);
  }

  seq{
   ReadGl3All();
  }
 }

 UpdateLoopCounters();
 if(iter == numIter ){
  par{
   runReadOpLoop = 0;
   runMainLoop = 0;
  }
 } else {
  delay;
 }
}

void WriteWfvPiped(unsigned 20 offset, unsigned 16 s,
     f_real *rout, unsigned 4 *rIndex, f_real *rReg,
     f_real *iout, unsigned 4 *iIndex, f_real *iReg){

 unsigned 1 WriteWfvIssueAddr, WriteWfvIssueData,
WriteWfvReadFromRam;
 unsigned 4 WriteWfvIndex;
 unsigned 32 WriteWfvIssueDataOut;
 unsigned 20 WriteWfvAddressSD;
 unsigned 4 WriteWfvCountSD;

 unsigned 1 tempBank;

 par{
  //reuse address calculator from readWfv routines
  calcWfvAddressStage1(s);
  WriteWfvLoop = 1;
  WriteWfvCountSD = 0;
  *rIndex = 0;
  *iIndex  = 0;
  WriteWfvIssueAddr = 1;
  WriteWfvIssueData = 0;
 }

 WriteWfvAddressSD = calcWfvAddressStage2(offset);

 do{
  par{
   if(WriteWfvCountSD == 11){
    WriteWfvIssueAddr = 0;
   } else { delay; }

   WriteWfvReadFromRam = WriteWfvIssueAddr;
   WriteWfvIssueData = WriteWfvReadFromRam;
   WriteWfvCountSD++;

   if(WriteWfvIssueAddr){
    par{
     WriteWfvAddressSD++;
     zbt2_a = WriteWfvAddressSD;
     zbt2_c = 0;

     zbt3_a = WriteWfvAddressSD;
     zbt3_c = 0;

     zbt4_a = WriteWfvAddressSD;
     zbt4_c = 0;

     zbt5_a = WriteWfvAddressSD;
     zbt5_c = 0;
    }
   } else {
    par{
     zbt2_c = 48;
     zbt3_c = 48;
     zbt4_c = 48;
     zbt5_c = 48;
    }
   }

   if(WriteWfvReadFromRam){
    par{
     delay;
     /*
     *rIndex = *rIndex+1;
     *iIndex = *iIndex+1;

     *rReg = rout[*rIndex];
     *iReg = iout[*iIndex];
     */
    }
   } else {delay;}

   if(WriteWfvIssueData){
    par{
     *rIndex = *rIndex+1;
     *iIndex = *iIndex+1;

     zbt2_oe = 1;
     zbt2_d = rout[*rIndex]\\32;

     zbt3_oe = 1;
     zbt3_d = rout[*rIndex]<-32;

     zbt4_oe = 1;
     zbt4_d = iout[*rIndex]\\32;

     zbt5_oe = 1;
     zbt5_d = iout[*rIndex]<-32;
          }
   } else {delay;}
  }
 }while(WriteWfvReadFromRam | WriteWfvIssueData);
}

void HLatMulM5Wfv_Stage3(unsigned 20 offset){
 WriteWfvPiped(offset, rSite_Stage3,
     xLatR, &xLatR_Index, &xLatR_Out,
     xLatI, &xLatI_Index, &xLatI_Out);
}

#ifdef SIMULATE
chanin unsigned 64 gIn with
  {infile = "g.txt", base = 16};
chanin unsigned 64 rIn with
  {infile = "rIn.txt", base = 16};
chanin unsigned 64 xIn with
  {infile = "xIn.txt", base = 16};
chanin unsigned 64 pIn with
  {infile = "pIn.txt", base = 16};
chanin unsigned 64 kappaIn with
  {infile = "kappa.txt", base = 16};
chanin unsigned 32 nsIn with
  {infile = "ns.txt", base = 10};
```

```
chanin unsigned 64 thresholdIn with
  {infile = "threshold.txt", base = 16};


unsigned 1 return_value;

unsigned 1 WriteVectorToSRAM(unsigned 20 startAddress, chanin f_real
(*ch), unsigned NSWIDTH NS_Val){
 f_real tempR, tempI;
 f_real rReg, iReg;
 ram f_real rWfv[12], iWfv[12];
 unsigned 20 i;
 unsigned 4 index;
 unsigned 4 rIndex, iIndex;
 unsigned 1 runLoop, alt;

 par{
  index = 0;
  i = 0;
 }

 while((i<-NSWIDTH) < (NS_Val)){

  while(index<12){
   *ch ? tempR;
   par{
    rWfv[index] = tempR;
    *ch ? tempI;
   }
   par{
    iWfv[index] = tempI;
    index++;
   }
  }

  WriteWfv(startAddress, i, rWfv, &rIndex, &rReg, iWfv, &iIndex,
&iReg);

  par{
   index = 0;
   i++;
  }
 }

 return 1;
}

macro proc ZeroGl3(gl3, i){
 i=0;
 do{
  par{
   gl3[i] = 0;
   i++;
  }
 }while(i<9);
}

unsigned 1 WriteMatrixToSRAM(unsigned 20 startAddress, chanin f_real
(*ch), unsigned NSWIDTH NS_Val){
 f_real tempR, tempI;
 f_real rReg, iReg;
 ram f_real rGl3[9], iGl3[9];
 unsigned NSWIDTH i;
 unsigned 3 j;
 unsigned 4 indexR, indexI, index;
 unsigned 4 rIndex, iIndex;
 unsigned 1 runLoop, alt;

 par{
  i = 0;
  index = 0;
  j = 0;
 }

 while(i < (NS_Val)){
  while(j < 4){
   while(index<9){
    *ch ? tempR;
    par{
     rGl3[index] = tempR;
     *ch ? tempI;
    }
    par{
     iGl3[index] = tempI;
     index++;
    }
   }

   WriteGl3(startAddress, (unsigned 20)(0 @ i),
    j<-2, rGl3, &rIndex, &rReg, iGl3, &iIndex, &iReg);

   par{
    index = 0;
    j++;
   }
  }
  par{
   i++;
   j = 0;
  }
 }

 return i<-1;
}

unsigned 32 RIDNS_Val;

void ReadInDataToSRAM(){
 f_real readTemp;
 unsigned 32 nsTemp;
 unsigned 20 pAddr;

 RIDNS_Val = 1;
 pAddr = PARMS_BASE;

 kappaIn ? readTemp;
 WriteBank2(pAddr, readTemp<-32);
 pAddr++;
 WriteBank2(pAddr, readTemp\\32);

 pAddr++;
 nsIn ? nsTemp;
 RIDNS_Val *= nsTemp;
 WriteBank2(pAddr, nsTemp);

 pAddr++;
 nsIn ? nsTemp;
 RIDNS_Val *= nsTemp;
 WriteBank2(pAddr, nsTemp);

 pAddr++;
 nsIn ? nsTemp;
 RIDNS_Val *= nsTemp;
 WriteBank2(pAddr, nsTemp);

 pAddr++;
 nsIn ? nsTemp;
 RIDNS_Val *= nsTemp;
 WriteBank2(pAddr, nsTemp);

 //set number of iterations, always one for simulation
 pAddr++;
 nsTemp = 1;
 WriteBank2(pAddr, nsTemp);

 thresholdIn ? readTemp;
 pAddr++;
 WriteBank2(pAddr, readTemp<-32);
 pAddr++;
 WriteBank2(pAddr, readTemp\\32);

 return_value = WriteVectorToSRAM
               (P_OFFSET, &pIn, RIDNS_Val<-14);
 return_value = WriteVectorToSRAM
                (R_OFFSET, &rIn, RIDNS_Val<-14);
 return_value = WriteVectorToSRAM
                (X_OFFSET, &xIn, RIDNS_Val<-14);

 return_value = WriteMatrixToSRAM
                (G_OFFSET, &gIn, RIDNS_Val<-14);
}

chanout unsigned 64 pOut with
  {outfile = "pOut.txt", base = 16};
chanout unsigned 64 rOut with
  {outfile = "rOut.txt", base = 16};
chanout unsigned 64 xOut with
  {outfile = "xOut.txt", base = 16};
chanout unsigned 64 tmp1Out with
  {outfile = "tmp1Out.txt", base = 16};
chanout unsigned 64 tmp2Out with
  {outfile = "tmp2Out.txt", base = 16};
chanout unsigned 64 othersOut with
  {outfile = "others.txt", base = 16};

unsigned 1 WriteMatrixToFile(unsigned 20 startAddress,
  chanout unsigned 64 (*ch), unsigned NSWIDTH NS_Val){
```

```
 unsigned NSWIDTH i;
 unsigned 3 j;
 f_real rReg, iReg;
 ram f_real tempR[9], tempI[9];
 unsigned 4 indexR, indexI, index;

 par{
  i = 0;
  index = 0;
 }

 while(i<NS_Val){
  while(j<4){
   ReadGl3(startAddress, (unsigned 20)(0 @ i), j<-2,
    tempR, &indexR, &rReg, tempI, &indexI, &iReg);

   while(index < 9){
    par{
     rReg = tempR[index];
     iReg = tempI[index];
    }
    *ch ! rReg;
    *ch ! iReg;

    index++;
   }
   par{
    j++;
    index = 0;
   }
  }
  par{
   j=0;
   i++;
  }
 }
 return i<-1;
}

unsigned 1 WriteVectorToFile(unsigned 20 startAddress,
  chanout unsigned 64 (*ch), unsigned NSWIDTH NS_Val){

 unsigned NSWIDTH i;
 f_real rReg, iReg;
 ram f_real tempR[12], tempI[12];
 unsigned 4 indexR, indexI, index;

 par{
  i = 0;
  index = 0;
 }

 while(i<NS_Val){
  ReadWfv(startAddress, (unsigned 20)(0 @ i), tempR,
   &indexR, &rReg, tempI, &indexI, &iReg);

  while(index < 12){
   par{
    rReg = tempR[index];
    iReg = tempI[index];
   }
   *ch ! rReg;
   *ch ! iReg;

   index++;
  }
  par{
   i++;
   index = 0;
  }
 }

 return i<-1;
}

void WriteOutResult(){

 return_value = WriteVectorToFile
                 (P_OFFSET, &pOut, RIDNS_Val<-14);
 return_value = WriteVectorToFile
                 (R_OFFSET, &rOut, RIDNS_Val<-14);
 return_value = WriteVectorToFile
                 (X_OFFSET, &xOut, RIDNS_Val<-14);
 return_value = WriteVectorToFile
               (TMP1_OFFSET, &tmp1Out, RIDNS_Val<-14);
 return_value = WriteVectorToFile
                (TMP2_OFFSET, &tmp2Out, RIDNS_Val<-14);

 othersOut ! res_old;
 othersOut ! alpha;
 othersOut ! beta;
 othersOut ! res_new;
}

#endif
}
```

D.5 Types.hch

```
typedef unsigned 64     f_real;

typedef f_real    half_gl3[9];
typedef f_real    half_wfv[12];

typedef f_real    ram half_gl3_ram[9];
typedef f_real    ram half_wfv_ram[12];

typedef mpram hlf_gl3_mpram
{
 ram f_real write[9];
 rom f_real read[9];
} half_gl3_mpram;

typedef mpram hlf_wfv_mpram
{
 ram f_real write[12];
 rom f_real read[12];
} half_wfv_mpram;

mpram half_gl3_mpram_block {
 ram f_real write[18];
 ram f_real readWrite[18];
};

mpram half_wfv_mpram_block
{
 ram f_real write[12];
 ram f_real read[12];
};

mpram dotRAM{
 ram f_real write[9];
 rom f_real read[9];
};

typedef mpram hlf_a_s_mpram
{
 ram unsigned 16 write[12];
 ram unsigned 16 readWrite[12];

}half_a_s_mpram;

typedef struct a_s_half_16bitRams
{
 half_a_s_mpram HH with {block = "BlockRAM"};
 half_a_s_mpram HL with {block = "BlockRAM"};
 half_a_s_mpram LH with {block = "BlockRAM"};
 half_a_s_mpram LL with {block = "BlockRAM"};

} a_s_ram;

typedef struct a_s_16bitRegisters
{
 unsigned 16 HH;
 unsigned 16 HL;
 unsigned 16 LH;
 unsigned 16 LL;
} a_s_reg;

typedef struct a_gl3
{
 half_gl3   r;
 half_gl3   i;
} f_gl3;

typedef struct b_gl3
{
 half_gl3_ram     r;
 half_gl3_ram     i;
} f_gl3_ram;

typedef struct c_gl3
{
 mpram half_gl3_mpram   r;
 mpram half_gl3_mpram   i;
} f_gl3_mpram;

typedef struct a_wfv
{
 half_wfv   r;
 half_wfv   i;
} f_wfv;

typedef struct b_wfv
{
 half_wfv_ram     r;
 half_wfv_ram     i;
} f_wfv_ram;

typedef struct c_wfv
{
 mpram half_wfv_mpram   r;
 mpram half_wfv_mpram   i;
} f_wfv_mpram;

macro proc Int2FP(a, r);

macro expr FP2Int(a);
```

D.6 Variables.hch

```
f_real alpha, alpha_minus, beta, res_old, res_new;
f_real dot_res, checkConjCond;

unsigned XWIDTH NX, NY, NZ, NT;
unsigned NSWIDTH NS;

f_real kappa, threshold;

unsigned 16 smx, spx;
unsigned 16 smy, spy;
unsigned 16 smz, spz;
unsigned 16 smt, spt;
unsigned 16 rSite, rSite_Result, rSite_Stage1;
unsigned 16 rSite_Stage2, rSite_Stage3;

unsigned XWIDTH x, xp1, xm1, NX;
unsigned YWIDTH y, yp1, ym1, NY;
unsigned ZWIDTH z, zp1, zm1, NZ;
unsigned TWIDTH t, tp1, tm1, NT;

unsigned NSWIDTH NS;

unsigned 32 numIter, iter;
unsigned NSWIDTH numNS;

half_wfv_ram xLatR, xLatI;
unsigned 4 xLatR_Index, xLatI_Index;
f_real xLatR_In, xLatR_Out, xLatI_In, xLatI_Out;

f_real a_spxR_Write_In, a_spxI_Write_In;
a_s_reg a_spxR_Write_Out, a_spxI_Write_Out, a_spxR_Read_Out,
a_spxI_Read_Out;

f_real gLat0R_In, gLat0I_In, gLat0R_Out, gLat0I_Out;

f_real ga_spxR_Write_In, ga_spxI_Write_In, ga_spxR_Write_Out,
ga_spxI_Write_Out;
f_real ga_spxR_Read_Out, ga_spxI_Read_Out;

//G5pG5Gx & MulGl3dWfv
half_wfv_mpram yLat1R, yLat1I;
a_s_ram a_smxR, a_smxI;
half_wfv_mpram    ga_smxR, ga_smxI with
  {block = "BlockRAM"};
mpram half_gl3_mpram_block gLat1R, gLat1I with
  {block = "BlockRAM"};

f_real a_smxR_Write_In, a_smxI_Write_In;
a_s_reg a_smxR_Write_Out, a_smxI_Write_Out, a_smxR_Read_Out,
a_smxI_Read_Out;

f_real gLat1R_In, gLat1I_In, gLat1R_Out, gLat1I_Out;

f_real ga_smxR_Write_In, ga_smxI_Write_In;
f_real ga_smxR_Write_Out, ga_smxI_Write_Out;
f_real ga_smxR_Read_Out, ga_smxI_Read_Out;

//G5mG5Gy & MulGl3Wfv
half_wfv_ram      yLat2R, yLat2I;
a_s_ram a_spyR, a_spyI;
half_wfv_mpram    ga_spyR, ga_spyI with
  {block = "BlockRAM"};
mpram half_gl3_mpram_block gLat2R, gLat2I with
  {block = "BlockRAM"};

a_s_reg a_spyR_Write_Out, a_spyI_Write_Out;
a_s_reg a_spyR_Read_Out, a_spyI_Read_Out;

f_real gLat2R_In, gLat2I_In, gLat2R_Out, gLat2I_Out;

f_real ga_spyR_Write_In, ga_spyI_Write_In;
f_real ga_spyR_Write_Out, ga_spyI_Write_Out;
f_real ga_spyR_Read_Out, ga_spyI_Read_Out;

//G5pG5Gy & MulGl3dWfv
half_wfv_ram yLat3R, yLat3I;
a_s_ram a_smyR, a_smyI;
half_wfv_mpram ga_smyR, ga_smyI with
  {block = "BlockRAM"};
mpram half_gl3_mpram_block gLat3R, gLat3I with
  {block = "BlockRAM"};

a_s_reg a_smyR_Write_Out, a_smyI_Write_Out, a_smyR_Read_Out,
a_smyI_Read_Out;

f_real gLat3R_In, gLat3I_In, gLat3R_Out, gLat3I_Out;

f_real ga_smyR_Write_In, ga_smyI_Write_In, ga_smyR_Write_Out,
ga_smyI_Write_Out;
f_real ga_smyR_Read_Out, ga_smyI_Read_Out;

//G5mG5Gz & MulGl3Wfv
half_wfv_mpram yLat4R, yLat4I;
a_s_ram a_spzR, a_spzI;
half_wfv_mpram    ga_spzR, ga_spzI with
  {block = "BlockRAM"};
mpram half_gl3_mpram_block gLat4R, gLat4I with
  {block = "BlockRAM"};

a_s_reg a_spzR_Write_Out, a_spzI_Write_Out, a_spzR_Read_Out,
a_spzI_Read_Out;
f_real a_spzR_Write_In, a_spzI_Write_In;

f_real gLat4R_In, gLat4I_In, gLat4R_Out, gLat4I_Out;

f_real ga_spzR_Write_In, ga_spzI_Write_In, ga_spzR_Write_Out,
ga_spzI_Write_Out;
f_real ga_spzR_Read_Out, ga_spzI_Read_Out;

//G5pG5Gz & MulGl3dWfv
half_wfv_mpram yLat5R, yLat5I;
a_s_ram a_smzR, a_smzI;
half_wfv_mpram ga_smzR, ga_smzI with
```

```
  {block = "BlockRAM"};
mpram half_gl3_mpram_block gLat5R, gLat5I with
  {block = "BlockRAM"};

a_s_reg a_smzR_Write_Out, a_smzI_Write_Out, a_smzR_Read_Out,
a_smzI_Read_Out;
f_real a_smzR_Write_In, a_smzI_Write_In;

f_real gLat5R_In, gLat5I_In, gLat5R_Out, gLat5I_Out;

f_real ga_smzR_Write_In, ga_smzI_Write_In, ga_smzR_Write_Out,
ga_smzI_Write_Out;
f_real ga_smzR_Read_Out, ga_smzI_Read_Out;

//G5mG5Gt & MulGl3Wfv
half_wfv_ram yLat6R, yLat6I ;
a_s_ram a_sptR, a_sptI;
half_wfv_mpram ga_sptR, ga_sptI with
  {block = "BlockRAM"};
mpram half_gl3_mpram_block gLat6R, gLat6I with
  {block = "BlockRAM"};

a_s_reg a_sptR_Write_Out, a_sptI_Write_Out, a_sptR_Read_Out,
a_sptI_Read_Out;

f_real gLat6R_In, gLat6I_In, gLat6R_Out, gLat6I_Out;

f_real ga_sptR_Write_In, ga_sptI_Write_In, ga_sptR_Write_Out,
ga_sptI_Write_Out;
f_real ga_sptR_Read_Out, ga_sptI_Read_Out;

//G5pG5Gt & MulGl3dWfv
half_wfv_ram yLat7R, yLat7I;
a_s_ram a_smtR, a_smtI;
half_wfv_mpram ga_smtR, ga_smtI with
  {block = "BlockRAM"};
mpram half_gl3_mpram_block gLat7R, gLat7I with
  {block = "BlockRAM"};

a_s_reg a_smtR_Write_Out, a_smtI_Write_Out, a_smtR_Read_Out,
a_smtI_Read_Out;

f_real gLat7R_In, gLat7I_In, gLat7R_Out, gLat7I_Out;

f_real ga_smtR_Write_In, ga_smtI_Write_In, ga_smtR_Write_Out,
ga_smtI_Write_Out;
f_real ga_smtR_Read_Out, ga_smtI_Read_Out;

//G5 rams and variables
half_wfv_ram yLat8R, yLat8I;
```

Appendix E

Layout on FPGA of double precision Dirac operator

Figure 2-a shows that FPGA layout of the double precision Dirac operator. The image shows where the logic for different parts of the design is placed in the FPGA. The image was obtained from the Xilinx FloorPlannner tool. The blocks of colour each represent a separate arithmetic unit, whilst the light green colour spread over the whole FPGA is the application logic that uses these arithmetic units. Any grey areas represent FPGA logic that is not used in the design.

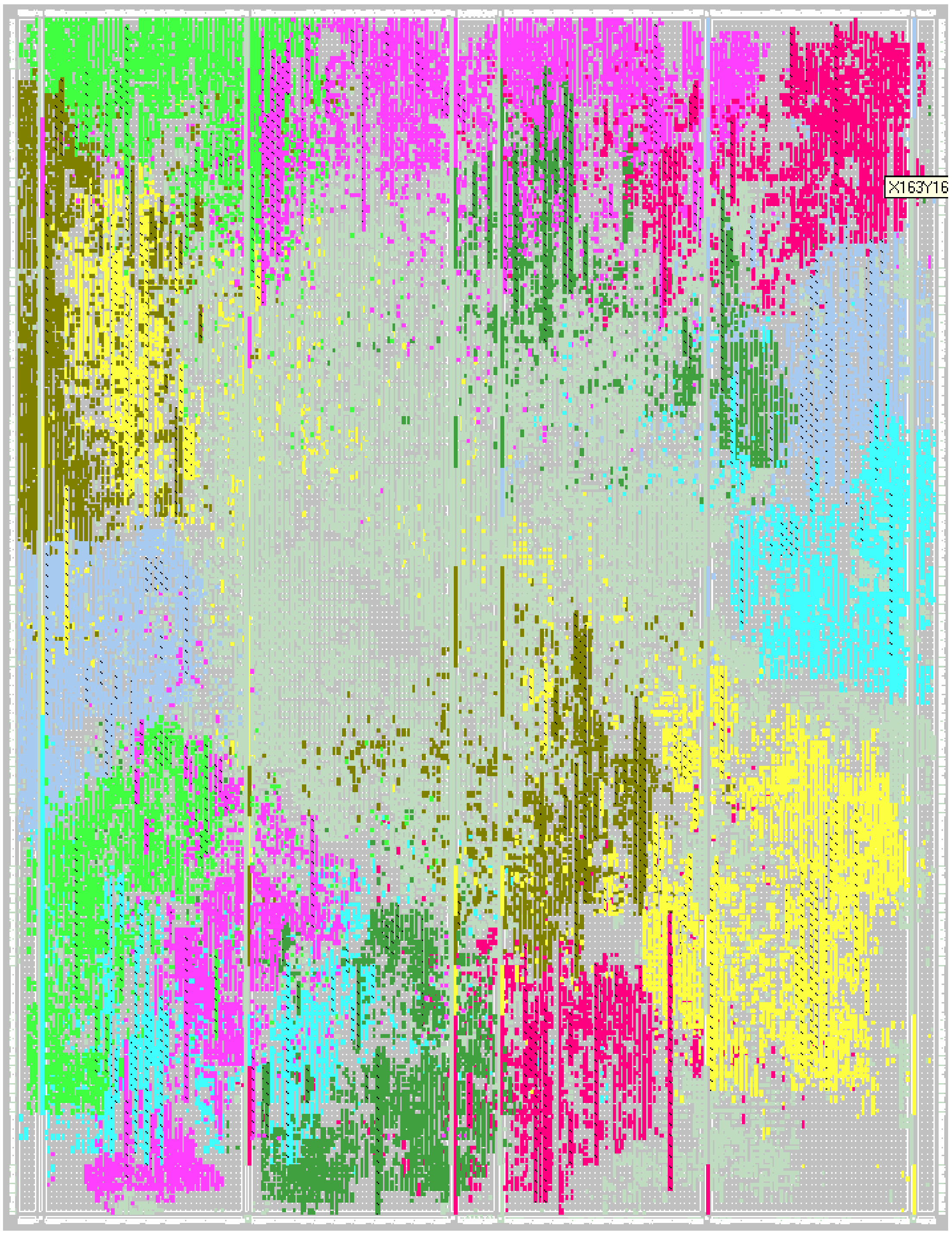


Figure 9-a. Layout of FPGA when programmed with the double precision Dirac operator design from Chapter 6. Each of the blocks of colour is one of the arithmetic units, the light green colour is the general application logic. Any grey areas represent logic that is not used in the design.

References

[A. Gara '05] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken and a. P. Vranas. Overview of the Blue Gene/L system architecture. IBM Journal of Research and Development, 49 195-212, 2005.

[Alpha-Data '05] Alpha-Data. Alpha-Data ADM-XRC-II Users Guide. 2005. http://www.alpha-data.co.uk/adm-xrc-ii.html

[Ammendola '05] R. Ammendola, M. Guagnelli, G. Mazza, F. Palombi, R. Petronzio, D. Rossetti, A. Salamon and P. Vicini. APENet: LQCD clusters a la APE. Nuclear Physics B - Proceedings Supplements, 140 826-828, 2005.

[Barrett '94] R. a. B. Barrett, M. and Chan, T. F. and Demmel, J. and Donato, J. M. and Dongarra, Jack and Eijkhout, V. and Pozo, R. and Romine, C. and der Vorst, H. V. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. Philadalphia: Society for Industrial and Applied Mathematics. Also available as postscript file on http://www.netlib.org\/templates\/Templates.html, 1994.

[Beauchamp '06a] M. J. Beauchamp, S. Hauck, K. D. Underwood and K. S. Hemmert. Embedded floating-point units in FPGAs. Proceedings of the international symposium on Field programmable gate arrays, 2006a.

[Beauchamp '06b] M. J. H. Beauchamp, Scott;, Underwood, Keith; Hemmert, K Scott;. Architectural Modifications to Improve Floating-Point Efficiency in FPGAs. 16th Int Conf on Field Programmable Logic and Applications, 515-520, 2006b.

[Belanovic '02] P. L. Belanovic, Miriam;. A Library of Parameterized Floating-Point Modules and Their Use. Field-Programmable Logic and Applications. Reconfigurable Computing Is Going Mainstream 12th International Conference, FPL 2002 657, 2002.

[Belletti '06] F. Belletti, S. F. Schifano, R. Tripiccione, F. Bodin, P. Boucaud, J. Micheli, O. Pene, N. Cabibbo, S. de Luca, A. Lonardo, D. Rossetti, P. Vicini, M. Lukyanov, L. Morin, N. Paschedag, H. Simma, V. Morenas, D. Pleiter and F. Rapuano. Computing for LQCD: apeNEXT. Computing in Science & Engineering, 8

18-29, 2006.

[Bhanot '05] G. Bhanot, D. Chen, A. Gara, J. Sexton and P. Vranas. QCD on the BlueGene/L Supercomputer. Nuclear Physics B - Proceedings Supplements, 140 823-825, 2005.

[Callanan '05] O. Callanan, A. Nisbet, E. Ozer, J. Sexton and D. Gregg. FPGA Implementation of a Lattice Quantum Chromodynamics Algorithm Using Logarithmic Arithmetic. Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International, 146b-146b, 2005.

[Callanan '06] O. Callanan, Peardon Mike, Nisbet Andy, Gregg David High performance scientific computing using FPGAs with IEEE floating point and logarithmic arithmetic for lattice QCD. 16 Int Conf on Field Programmable Logic and Applications, 2006.

[Celoxica '06] Celoxica. Celoxica RC2000 development board. 2006. http://www.celoxica.com/products/rc2000/default.asp

[Coleman '99] J. N. Coleman and E. I. Chester. A 32 bit logarithmic arithmetic unit and its performance compared to floating-point. 142-151, 1999.

[Coleman '00] J. N. Coleman, E. I. Chester, C. I. Softley and J. Kadlec. Arithmetic on the European logarithmic microprocessor. Computers, IEEE Transactions on, 49 702-715, 2000.

[Combet '65] M. V. Z. Combet, H;Verbeek, L;. Computation of the Base Two Logarithm of Binary Numbers. IEEE Trans. Electronic Computers, vol. EC-14 863-867, 1965.

[Davies '98] C. Davies. Let's Play Quantum Chess. New Scientist, 2137 pp 32-35 6 June 1998

[Davies '03] C. Davies. Teraflop computing tackles the strong force. Frontiers, 22-24, 2003.

[Davies '00] C. C. Davies, Sara;. Physicists get to grips with the strong force. Physics WOrld, 8 pp August 2000 2000

[Dou '05] Y. Dou, S. Vassiliadis, G. K. Kuzmanov and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays, 2005.

[Ellis '86] J. R. Ellis. Bulldog : a compiler for VLIW architectures. ACM doctoral dissertation award ; 1985, MIT Press, 1986.

[Fagin '94] B. Fagin and C. Renard. Field programmable gate arrays and floating point arithmetic. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 2 365-367, 1994.

[Gellrich '03] A. e. a. Gellrich. Lattice QCD calculations on commodity clusters at DESY. Proceedings of Computing in High Energy Physics 2003. Publushed by eConf ref C0303241., 2003.

[Gottlieb '01] S. Gottlieb. Comparing clusters and supercomputers for lattice QCD. Nuclear Physics B - Proceedings Supplements, 94 833-840, 2001.

[Govindu '02] G. Z. Govindu, Ling; Choi, S; Gundala, Padma; Prasanna, Viktor K;. Area and Power Performance Analysis of a Floating-point based Application on FPGAs. Seventh Annual Workshop on High Performance Embedded Computing, HPEC 2002, 2002.

[Hall '70] E. L. Hall, D. D. Lynch and S. J. Dwyer. Generation of Products and Quotients Using Approximate Binary Logarithms for Digital Filtering Applications. Computers, IEEE Transactions on, C-19 97-105, 1970.

[Haselman '05] M. Haselman, M. Beauchamp, A. Wood, S. Hauck, K. Underwood and K. S. Hemmert. A comparison of floating point and logarithmic number systems for FPGAs. Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on, 181-190, 2005.

[Hennessy '90]J. L. Hennessy, D. A. Patterson and D. Goldberg. Computer architecture : a quantitative approach. Morgan Kaufman Publishers, 1990.

[Hoare '85] C. A. R. Hoare. Communicating sequential processes. Prentice-Hall International, 1985.

[Holmgren '05a] D. Holmgren. Cluster Development at Fermilab. 2005a. http://lqcd.fnal.gov/allhands_holmgren.pdf

[Holmgren '05b] D. J. Holmgren. PC Clusters for Lattice QCD. Nuclear Physics B - Proceedings Supplements, 140 183-189, 2005b.

[Holmgren '06] D. M. Holmgren, Paul; Simone, Jim; Singh, Amitoj;. Lattice QCD Clusters at Fermilab. Computing in High Energy Physics, 2006.

[IEEE '85] IEEE. IEEE Standard for binary floating-point arithmetic. 1985.

[IEEE '99] IEEE. Information technology - telecommunications and information exchange between systems - local and metropolitan area networks - specific requirements. Supplement to Carrier Sense Multiple Access with Collision Detection (CSMA/CD) access method and physical layer specifications - physical layer parameters and specifications for 1000 Mb/s operation over 4-pair of category 5 balanced copper cabling, type 1000BASE-T. IEEE Std 802.3ab-1999, 1999.

[Infiniband '06] Inifiband Technology Overview;

http://www.infinibandta.org/about/
[INMOS. '84] INMOS. OCCAM programming manual. Prentice-Hall International, 1984.

[JEDEC '05] JEDEC. JEDEC Standard 79, Double Data Rate (DDR) SDRAM Specification. 2005. http://www.jedec.org/

[Kingsbury '71] N. R. Kingsbury, PJW;. Digital FIltering Using Logarithmic Arithmetic. Electronics Letters, 7 56-58, 1971.

[Koren '93] I. Koren. Computer arithmetic algorithms. Prentice-Hall ; Prentice-Hall International (UK), 1993.

[Kurokawa '80] T. Kurokawa, J. Payne and S. Lee. Error analysis of recursive digital filters implemented with logarithmic number systems. Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing], IEEE Transactions on, 28 706-715, 1980.

[Ligon '98] W. B. Ligon, III, S. McMillan, G. Monn, K. Schoonover, F. Stivers and K. D. Underwood. A re-evaluation of the practicality of floating-point operations on FPGAs. 206-215, 1998.

[Logue '05] J. Logue. Virtex-II SelectLink Communications Channel, XAPP263. Xilinx Application Notes, 2005.
[Louca '96] L. Louca, T. A. Cook and W. H. Johnson. Implementation of IEEE single precision floating point addition and multiplication on FPGAs. 107-116, 1996.

[Luscher '02] M. Luscher. Lattice QCD on PCs? Nuclear Physics B - Proceedings Supplements, 106-107 21-28, 2002.

[Matousek '02] R. Matousek, M. Tichy, Z. Pohl, J. Kadlec, C. Softley and N. Coleman. Logarithmic Number System and Floating-Point Arithmetics on FPGA. 12th International Conference on Field Programmable Logic and Applications. LNCS Vol 2438., 627, 2002.

[Matousek '03] European Logarithmic Microprocessor Project; http://www.utia.cas.cz/ZS/projects/hsla/elm.pdf
[Mitchell '62] J. Mitchell. Computer Multiplication and Division using Binary Logarithms. IEEE Transactions on Electronic Computers, vol.EC- 11 512-517, 1962.

[Moloney '04] D. Moloney, D. Geraghty and F. Connor. The performance of IEEE floating-point operators on FPGAs. Irish Signals and Systems Conference, 601-606, 2004.

[OpenCores '06] OpenCores Initiative; http://www.opencores.org
[P A. Boyle '05] P A. Boyle, D. Chen, N H. Christ, M A. Clark, S. D. Cohen, C. Cristian, Z. Dong, A. Gara, B. Joó, C. Jung, C. Kim, L. A. Levkova, X. Liao, G. Liu, R. D. Mawhinney, S. Ohta, K. Petrov, T. Wettig and A. Yamaguchi. Overview

of the QCDSP and QCDOC computers. IBM Journal of Research and Development, 49 351, 2005.

[P. A. Boyle '05] P. A. Boyle, D. Chen, N. H. Christ, M. A. Clark, S. D. Cohen, C. Cristian, Z. Dong, A. Gara, B. Joó, C. Jung, C. Kim, L. A. Levkova, X. Liao, G. Liu, R. D. Mawhinney, S. Ohta, K. Petrov, T. Wettig and A. Yamaguchi. Overview of the QCDSP and QCDOC computers. IBM Journal of Research and Development, 49 351, 2005.

[Press '92] W. H. Press. Numerical recipes in C : the art of scientific computing. Cambridge University Press, 1992.

[Roesler '02] E. Roesler and B. Nelson. Novel Optimizations for Hardware Floating-Point Units in a Modern FPGA Architecture. Lecture Notes in Computer Science, 2002.

[Severance '98] C. Severance. IEEE 754: An Interview with William Kahan. IEEE Computer, 31 114-115, 1998.

[Shirazi '95] N. Shirazi, A. Walters and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), 155-162, 1995.

[Sicuranza '83] G. Sicuranza. On efficient implementations of 2-D digital filters using logarithmic number systems. Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing], IEEE Transactions on, 31 877-885, 1983.

[Swartzlander '75] E. E. Swartzlander, Jr. and A. G. Alexopoulos. The Sign/Logarithm Number System. Computers, IEEE Transactions on, C-24 1238-1242, 1975.

[Swartzlander '83] E. E. J. C. Swartzlander, D V Satish; Nagle, H Troy Jr; Starks, Scott A;. Sign/Logarithm Arithmetic for FFT Implementation. IEEE Trans. Computers, 32 526-534, 1983.

[Underwood '04a] K. Underwood. FPGAs vs. CPUs: trends in peak floating-point performance. Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, 2004a.

[Underwood '04b] K. D. Underwood and K. S. Hemmert. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. Field-Programmable Custom Computing Machines, 2005. FCCM 2004. 12th Annual IEEE Symposium on, 219-228, 2004b.

[Wettig '05] T. Wettig. Performance of Machines for Lattice QCD SImulations. Lattice 2005. Proceedings of Science ref. PoS(LAT2005)019., 2005.

[Xilinx '05a] Xilinx. Xilinx Virtex-II FPGA Datasheet. 2005a.
http://direct.xilinx.com/bvdocs/publications/ds031.pdf

[Xilinx '05b] Xilinx. Xilinx Virtex-II Pro Datasheet. 2005b.
http://direct.xilinx.com/bvdocs/publications/ds083.pdf

[Xilinx '06a] Xilinx. Xilinx Virtex-4 Family Overview. 2006a.
http://direct.xilinx.com/bvdocs/publications/ds112.pdf

[Xilinx '06b] Xilinx. Xilinx Virtex-5 Datasheet. 2006b.
http://direct.xilinx.com/bvdocs/publications/ds100.pdf

[Zhuo '04] L. Zhuo and V. K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on FPGAs. Proceedings of 18th International Parallel and Distributed Processing Symposium, 92, 2004.