Reconfigurable Microarchitectures at the Programmable Logic Interface

Adam Donlin

Doctor of Philosophy
University of Edinburgh
2001

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text. Some of the work also appears in the following papers:

- G. Brebner and A. Donlin. "Runtime Reconfigurable Routing". In José Rolim, editor, *Parallel and distributed Processing*, volume 1388 of *LNCS*, pages 25–30. Springer-Verlag, 1998.

- A. Donlin. "Self Modifying Circuitry - A Platform for Tractable Virtual Circuitry". In A. Keevallik, R. W. Hartenstein, editor, *Field Programmable Logic and Applications - From FPGAs to Computing Paradigm*, volume 1482 of *LNCS*, pages 199–208. Springer-Verlag, 1998.

Abstract

Dynamic, runtime reconfiguration is one of the most compelling, yet elusive applications of programmable logic. The lack of an accepted design methodology and limitations of the programmable logic interface are identified as two significant factors constraining the mainstream acceptance of runtime reconfiguration and virtual circuitry(VC). This thesis presents a framework for investigating a new form of flexible programmable logic interface capable of adapting to the demands of different VC models. An abstract architecture for virtual circuitry is presented in the context of two fundamental models of VC: the sea of accelerators and the parallel harness. The abstract architecture's position within the class of Transport Triggered Architectures(TTAs) is considered and we discuss how attributes of the architecture are harnessed to facilitate a third, sequential algorithmic VC model.

A novel implementation of the abstract architecture is described: the implementation of the Ultimate RISC(URISC), a minimal microarchitecture, is presented and is then evolved into the Flexible URISC(FURI), an instance of the abstract VC architecture. A design flow and associated toolset for the FURI core is presented. This includes a discussion of the merits and complications of different strategies for circuitry loading plus the features of a multitasking runtime environment for the FURI core, the FURI executive. Starting with the description of a simple base protocol, the design space for FURI protocols is qualified. The communication characteristics of the three VC models are described and their influence on the form of FURI protocols considered. Implementations of the Data Encryption Standard(DES) are proposed, demonstrating how the FURI system supports each of the three VC models.

To my parents.

Acknowledgements

Over the years, I have had the tremendous privilege of working with and being inspired by many wonderful people. I am grateful to each and every one, but I feel it is appropriate to offer special thanks to three of them in particular:

- First, I wish to thank Gordon Brebner for his support and supervision. His insightful guidance, patience, and overall dedication are what make him a truly great supervisor.

- Second, I wish to thank John Gray for many stimulating discussions. His verve, encouragement, and insights have inspired me many times whilst working on this thesis.

- Third, I wish to thank Ray Welland for his support and encouragement during the years I spent studying in Glasgow.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Runtime Reconfiguration

Dynamic, runtime reconfiguration is one of the most compelling and yet, at the same time, elusive applications of field programmable logic devices. Its main goal is to exact a higher degree of system performance by dynamically adapting the logic circuitry used within an application. This is done by realising at least part of a system as application-specific logic circuitry to be implemented on an FPGA. The re-programmability of FPGAs is then harnessed to dynamically instantiate, and then possibly specialise or alter, the logic circuitry as the computational demands of the application change.

In essence, runtime reconfiguration makes the traditional tradeoff between system performance and system flexibility much more fluid: it gives the application designer the opportunity to harness the low level parallelism of circuitry to gain performance while retaining flexibility through the dynamic instantiation of circuitry on the programmable logic array. Virtual circuitry is a technique that exploits dynamic reconfiguration to implement circuit swapping. Effectively, dynamic reconfiguration is used to facilitate the illusion of having a larger circuitry resource than is actually, physically available.

1.1.1 Challenges

The density and degree of programmability of FPGAs has improved over successive generations and this has done much to improve the tractability of virtual circuitry by, in theory, enabling more complex computational elements to be mapped onto reconfigurable logic. Despite these advances, runtime reconfiguration is still a delicate technique, typically applied in an ad hoc manner. Furthermore, just as the programmable logic world has evolved to become more flexible, general-purpose, so microprocessors have evolved to be increasingly parallel. Indeed, a case study [92] has shown that such advances in microprocessor architecture have been successful in recouping some of the readily available performance advantages demonstrated in early virtual circuitry systems with a comparable degree of design effort.

In 2000, there are two particularly notable challenges to the general deployment of runtime reconfiguration:

- first, there is no widely accepted design methodology that facilitates the design of runtime reconfigurable systems and, as a consequence, only very limited CAD support; and

- second, the interface to the programmable logic resource itself predominantly fails to adequately support runtime reconfiguration and virtual circuitry.

Furthermore, it is often the case that the typical system environment in which runtime reconfiguration, and virtual circuitry in particular, is deployed places the reconfigurable logic on the host system's peripheral bus and starves it of bandwidth. Whilst this is an effective means of introducing programmable logic into mainstream systems, the resulting architecture is not conducive to implementing rapid, tightly integrated runtime reconfiguration and virtual circuitry.

1.2 Aim of the thesis

In general, this thesis focuses on the second of the two challenges discussed above, although we may allude to aspects of design methodologies for runtime reconfiguration and tool support at various points. In a holistic sense, this thesis presents a framework for investigating a new form of flexible, adaptable programmable logic interface. In particular, we shall consider the nature of the programmable logic interface and discuss a challenging, novel implementation of an abstract, reconfigurable microarchitecture that has a unique relationship to its host FPGA and supports three major models of virtual circuitry.

1.3 Thesis Outline

Chapter 2 gives an overview of the history of programmable logic and programmable logic device architectures. In particular, the chapter describes those architectures that define the state of the art in mainstream, commercial FPGA architecture when this thesis was written in 2000. Particular attention is also given to the partially reconfigurable mainstream architectures that have shaped dynamic reconfiguration research.

Chapter 3 explores the concept of the programmable logic interface at its different abstractions within a dynamically reconfigurable system. Focusing down at the level of the programmable logic device interface, the chapter discusses how the programmable logic device interface has evolved from its early, serialised forms to the more advanced streaming, packet oriented device interfaces that are tailored to support particular application classes. The notion of a flexible programmable logic interface that is capable of adapting to the demands of different applications is then introduced.

Chapter 4 defines an abstract architecture supporting virtual circuitry. First, the two fundamental models of virtual circuitry are introduced, then the

form and semantics of the abstract architecture are presented. The discussion is broadened, briefly, to consider the architecture within the class of Transport-Triggered Architectures(TTAs). The discussion continues on to consider how the attributes of the abstract architecture are harnessed in a way that facilitates a third model of virtual circuitry.

Chapter 5 presents the implementation of the abstract microarchitecture introduced in the previous chapter. The design and operation of the Ultimate RISC (URISC) is presented in detail and is then evolved into the *Flexible* URISC (FURI). The main component of this chapter, therefore, is a detailed technical discussion of the FURI core and the unique challenges to the implementation of a self-modifying microarchitecture on the Xilinx XC6200 FPGA. Key features of the target FPGA architecture, such as the FastMap interface and an open configuration bitstream, are highlighted.

Chapter 6 presents the design flow, its associated toolset, and a runtime environment for the FURI core. This chapter pays particular attention to the merits and complications associated with different approaches to loading SLU bitstreams. The FURI executive is introduced as a basic runtime operating environment for the FURI core and the chapter concludes with a description of the base protocol used to communicate with the core.

Chapter 7 expands the discussion of FURI protocols and explores how the form of a protocol can be influenced by the communication characteristics of the three VC models and the particular FURI network architecture. The chapter concludes with a description of three proposed implementations of the Data Encryption Standard (DES) in each of the VC styles to demonstrate how the FURI system can support all three VC models.

Chapter 8 ends the thesis with a presentation of conclusions and suggested areas for future work.

Chapter 2

Reconfigurable Architectures and Systems

Programmable logic is a central enabling technology exploited by this thesis. This chapter has three aims related to programmable logic:

- the first aim is to present a short history of field programmable logic, discussing the key points in the evolution of the programmable machine, and by highlighting significant architectural research contributions;
- the second aim is to present the current state of the art in field programmable logic device architecture. This covers both contemporary commercial and research architectures;
- the third is to briefly introduce the two main classes of Reconfigurable Computing systems in which programmable logic devices and architectures are typically harnessed.

Field Programmable Logic Arrays are a new class of computational device. A clearer definition of what is meant by a programmable logic device is appropriate at this point.

2.1 History of the Programmable Machine

The microprocessor has held position as the dominant form of programmable computing device for the last 20 years and has a rich history of predecessors.

The first widely acknowledged programmable device was conceived by Charles Babbage in the early 1800s [66] but his efforts to build the ambitious "Analytical Engine" were confounded by the engineering limitations of the time. Approximately one hundred years later, however, Alan Turing introduced his model of a universal computing machine called, simply, the "Turing Machine". The distinguishing feature of Turing's machine was that it had a mathematically complete, underlying model of computation allowing it to implement any of the "decidable problems" [103]. So profound was this contribution that Turing's model is recognised as sparking the development of the modern electronic computer.

2.1.1 Evolution of the Microprocessor

In the early 1940s, J.P. Eckert and J. Mauchly developed the first electronic programmable device, called ENIAC. Their ideas were crystalised further by John von Neumann who suggested the model of the "stored-program" computer. This architecture was first realised in the Manchester Mark I [62] which is acknowledged as the first electronic computer to execute a stored program. In the fifty years that have followed, many different forms of electronic computer have been designed and built. The underlying electronic technologies have changed: vacuum tubes yielded to discrete transistors which, in turn, yielded to Integrated Circuits(ICs) which then yielded to Very Large Scale Integration(VLSI). With each change of technology, a new generation of computer has arisen but, despite these technological changes, the core of the majority of electronic computers developed has essentially remained the stored-program architecture of the 1940s. The microprocessor is a direct product of the VLSI generation of computer devices and is, essentially, a complete stored-program architecture in a single packaged IC. The millions of micro-scale transistors at the disposal of the VLSI designer enable the integration of such complex architectures. Indeed, Moore's law observes that, over the last 20 years, IC transistor feature size has, on average,

halved every eighteen months and provided designers with double the number of transistors to exploit. In 2000, the state of the art in integrated circuit technology has a transistor feature size of 0.13μ and is leading designers towards a new generation of computers implemented as deep-submicron ICs containing over a billion transistors. Exploiting such deep-submicron devices efficiently has become an important system architecture research challenge [18]. System-level integration is one approach to this problem, facilitating the integration of not just the microprocessor, but a complete system on a single chip [74].

2.1.2 Logic Circuitry

The essential programmability of the microprocessor lies in its ability to execute different sequences of a set of "core" instructions. These core instructions do not change and are effectively cast in stone within the physical design of the device itself. At this low level, however, the microprocessor is implemented as a set of digital logic circuits. The vast majority of microprocessors and computing devices designed in the last 50 years are digital devices. They harness a physical phenomenon, typically the flow of electrical current, and reduce the continuous nature of that phenomenon to a finite set of discrete states, typically two. Abstract values represented in the digital domain are then encoded as a sequence of discrete digital values. The programmable logic devices discussed in the next section are digital devices, although there are notable examples of analog and mixed-signal architectures [39].

Logic circuits are implementations of boolean logic expressions, constructed according to the principles of digital design. Digital design can be broadly classified in two categories, differentiated by the timing discipline they employ: that is, synchronous and asynchronous digital design. Both categories have advantages and disadvantages. Synchronous systems are widely regarded as simpler to design but the timing abstraction realised through the use of a global clock

creates inefficiencies in the system. Fast elements in the design are constrained to the timing flow of the slowest element in the design. On the other hand, asynchronous systems exploit the "natural" timing of components, and are efficient at the implementation level: the system as a whole reacts on a continuous timescale as opposed to the discrete, stepped nature of a synchronous design. This timing efficiency has a proportional effect on the speed and power consumption of the asynchronous system. The continuous, analogue timing discipline, however, means designers of asynchronous systems face a more difficult design task. Timing hazards and glitches, for example, must be explicitly managed in the design process. This thesis will consider mainly synchronous digital systems.

Synchronous digital design defines differing levels of abstractions to assist logic circuit design. At the lowest level, the switch level, the fundamental component in the implementation is a simple switch. The switch provides a physical basis for the two state digital system by either allowing or preventing the flow of current depending on the presence of an electrical charge at the switch's control line. An engineer may construct the physical analogue of boolean algebra expressions by constructing networks of interlinked switches.

The switch level is inconvenient for all but the smallest designs. At the next level of digital design abstraction, the logic level, switches are grouped into structures which are equivalent to familiar boolean logic operations. These structures, called logic gates, are then used as the fundamental design component. The interconnected network of logic gates forms a "logic circuit" which may then be hierarchically composed with other circuits to form increasingly complex systems. Logic and switch level circuits are inherently parallel entities. Whilst the user perceives the sequential execution of a sequence of instructions on a microprocessor, at the logic and switch levels, components are operating in parallel to implement a particular computation. This is the fundamental difference in the nature of logic devices and microprocessors. In general, the ability to exploit parallelism in an

implementation yields a faster computation. This is reflected in the advancement of microprocessor architectures in the last 20 years, where a gradual erosion of the purely sequential execution nature has occurred. Each new generation of microprocessor presents more and more of the underlying parallelism of their physical implementation to the programmer in an attempt to gain higher performance at the software level.

2.1.3 Programmable Logic

A logic device is the physical implementation of a logic circuit in a particular electronic technology. Whilst valve and discrete transistor implementations of logic devices are possible, it is really the advent of IC technology that has had the most significant impact on logic device density. To that effect, the first generation of commodity logic devices did not appear until the advent of small scale ICs in the late 1960s. These devices underpinned a whole generation of computers but three of their characteristics are notable: firstly, the devices had a small density in the order of tens of transistors; secondly, like all new technologies, logic devices were expensive to manufacture; and, finally, a logic device is a fixed purpose component. Unlike the preceding generations of electronic computer, which were capable of executing different sequences of instructions, a traditional logic device only ever implements one particular logic circuit. In essence, traditional logic devices are not programmable.

A programmable logic device is defined as a physical device which can be programmed to implement a variety of different logic circuits without the loss of inherent parallelism. This differs from a microprocessor emulation of a logic circuit which serialises the effect of each component in the circuit to fit the processor's sequential execution model.

Like the microprocessor, it is possible to identify different generations of programmable logic device, classifying them in relation to their architecture, degree of

re-programmability, and underlying semiconductor technology. Currently, three main generations of programmable logic can be discerned, each of which are discussed in the subsections below. Just as the visionary research contributions which influenced the development of the microprocessor and electronic computer are noted, the equivalent contributions which influenced the development of programmable logic before practical implementation was feasible can be discerned. These significant contributions are discussed below.

2.2 Early Forms of Programmable Logic

2.2.1 The Fixed-plus-Variable Structure (F+VS) Computer

The earliest notion of a custom-computing machine which supported alterations to its logic hardware configuration was proposed in Estrin's F+VS Computer [35]. Estrin's architecture was a stored program machine comprising a fixed part, in the form of a general purpose computer, and a variable part, in the form of an inventory of special-purpose circuit substructures. Substructures would be added or removed from the machine as a means of temporarily transforming that machine from a general-purpose computer, into a high-speed special-purpose computer. He proposed that, as the application demands varied with time, the set of substructures present in the machine could be altered to maintain the high-performance of a specialised architecture. Essentially, Estrin describes an early form of architecture supporting the philosophy of contemporary, programmable logic driven Custom Computing machines. Applications which could benefit from the flexible nature of the F+VS were even considered [36].

It is notable that, in the spirit of the F+VS machine, substructures are not simply peripheral devices: instead of being resident on an auxiliary bus or IO bus, substructures are have a close relationship to the fixed core of the machine. One question raised explicitly in the F+VS literature considers the extent to which substructures can interface to the fixed core – in particular, how much memory

should they directly share and to what degree can a substructure vary bit widths of its datapaths from the fixed core bitwidth?

Estrin's machine predated even the beginnings of the IC era and the physical mechanisms available for supporting a flexible machine architecture were not particularly elaborate. Substructures were Printed Circuit Board(PCB) daughtercards and altering the logical configuration of the machine would most likely require the physical installation (i.e. soldering) of the required substructures. These technological limitations of the time would prevent the rapid reconfiguration seen in later programmable logic devices, constraining the machine to being configured strictly on a per-application basis.

2.2.2 Cutpoint Cellular Logic

A cellular array is a geometric arrangement of homogeneous cells that are interconnected in some regular topology. The cells of the array perform some particular logical function and, in early arrays, that function was fixed by the physical design of the device. The interconnection of cells is dynamic and, using this property, a customised dataflow between cells can be constructed. Cascading data through cells allowed more complex logical functions to be computed. Later cellular arrays allowed both the interconnect and cell function to be customised so that, although the circuits fabricated were still homogeneous, each cell could implement a more flexible range of functions. The typical means of programming the arrays included physically blowing fuses to make 'cuts' at appropriate points in the cell circuitry, or implementing switches with photo-conductors.

Cellular logic devices appeared in the early 1960s [82]. The cellular techniques they embodied became popular as a means of exploiting the increasingly reliable batch-fabrication processes emerging at the same time. Designers were motivated to take advantage of the new fabrication processes to produce devices which were cheaper, smaller, and potentially more reliable. Notably, the architecture of

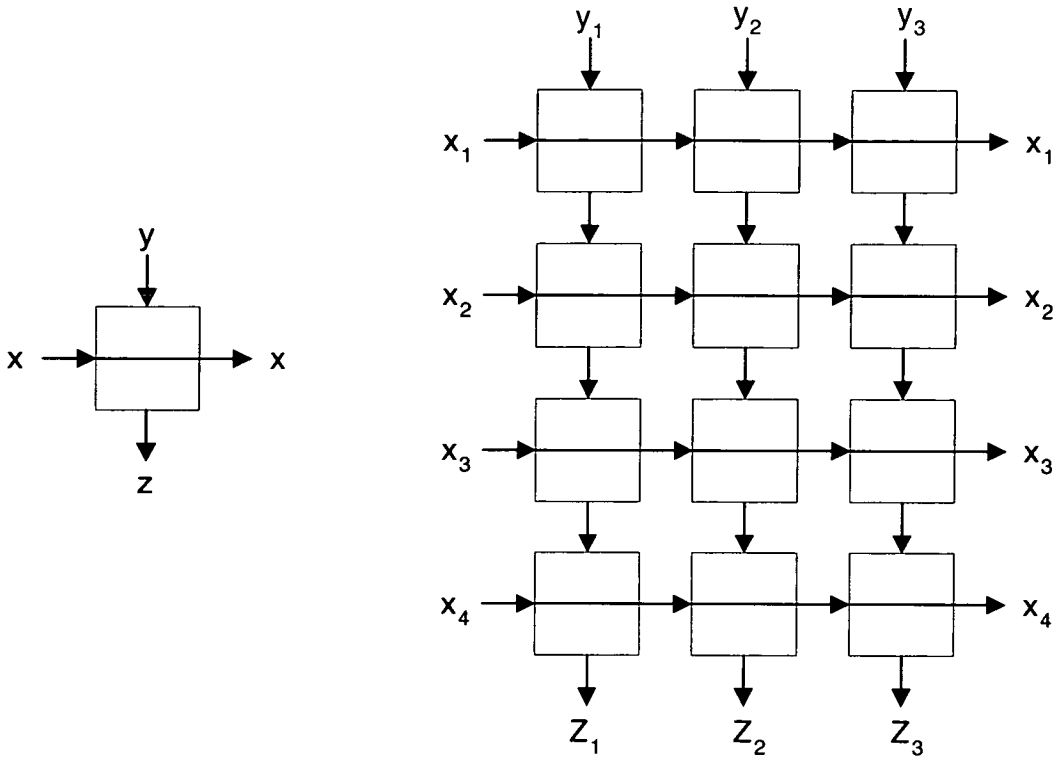

Figure 2.1: Minnick's Cutpoint Array

cellular arrays had a key role in increasing their reliability. Faults in the physical device could be tolerated by customising the flow of data around an affected cell or interconnect.

The architecture of cellular arrays makes them close, early relatives to devices from the second generation of programmable logic. Cutpoint Cellular Logic [81], for example, is a class of cellular array devices whose architecture influenced an important series of FPGAs. The basic architecture of the cutpoint array is a two-dimensional grid of cutpoint cells interconnected by directed, horizontal busses and directed, vertical cell-to-cell routes (this is shown in Figure 2.1). Each cell is specialised according to four bits and can implement one function from a set of 64, plus a reset-set flip-flop. Cutpoint arrays are derived from a Maitra cascade [73] in such a way that they are capable of implementing arbitrary functions of n inputs in a cutpoint array of $n-1$ cells high and no more than $2n-2$ cells wide.

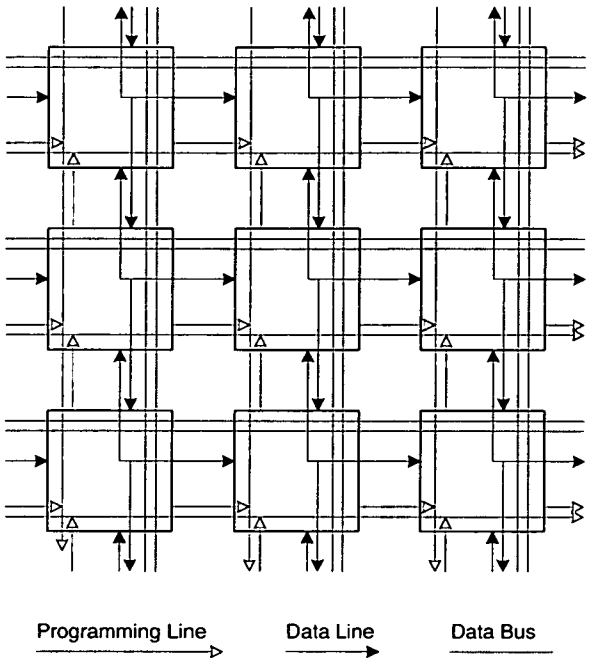

Figure 2.2: The Wahlstrom Programmable array

(The reader is referred to [81] for details of the limitations of the basic Maitra cascade and a discussion of how the cutpoint array was then derived.)

2.2.3 Wahlstrom's Programmable Logic Array

The Wahlstrom Array [108] also adopted the cellular logic array approach, but is notable for some of the architectural features it possessed. The cells of the Wahlstrom array were arranged in a conventional rectangular grid but had a mixture of interconnections allowing direct communication with the nearest neighbours of a cell in any compass direction and, additionally, non-adjacent cells could exploit a set of 'flight-lines' spanning the array in both the X and Y directions. The Wahlstrom architecture and cell structure is shown in Figure 2.2. The programming of the Wahlstrom array was particularly advanced for its time: each cell in the array had 13 control flip-flops that governed key switching points in the cell circuitry. The entire array would be programmed by loading data values into the control flip flops of each cell.

The Wahlstrom array is notable for the number of features it possessed that

are found in the FPGA devices developed some 25 years later. The use of re-programmable state elements to hold the configuration of a cell underpins the most successful generation of FPGA devices that use SRAM memories to hold cell and interconnect configurations. Also, the availability of both bussed and neighbour interconnects is similar to the complex routing structures available in contemporary FPGAs.

One particularly interesting feature of the Wahlstrom array was that each cell had access to the programming controls of the neighbour cells above and uncommitted to the right. Essentially one cell could reprogram a neighbour cell by writing values on the programming lines of that neighbour cell. As will be discussed in more detail in Chapter 4, allowing access to the programming interface from inside a programmable logic device is one of the key requirements for implementing self-modifying circuitry.

2.2.4 Shoup and the Programmable Cellular Logic Array

By 1970, a decade of research on cellular logic arrays had passed, and designers were on the eve of the widespread introduction of LSI fabrication technologies. At this point, Shoup presented a thesis [95] that forecast in detail the FPGA devices that were to be introduced some 15 years later. The main contributions of Shoup's thesis come from, firstly, his attempts to systematically assess cellular architectures along a variety of dimensions and, additionally, from the array architectures designed in relation to these assessments. He defines a number of different dimensions for this purpose that include generality, logical size, the array geometry, the cell functionality, the array's interconnection structure, and the number of state elements available per cell. By constructing metrics that are based on these dimensions, the thesis then considers the design of three programmable cell arrays: two for low-generality applications and one for high-generality applications.

The overall theme of the thesis is concerned with the details of array archi-

tecture design. That given, however, the thesis remains notable for making such early reference to aspects of programmable logic that have remained active research topics 15 years after the introduction of FPGAs. Topics equivalent to dynamic reconfiguration, and self-modifying circuitry, are given explicit mention and singled out as worthy areas of future research.

2.2.5 First Generation Programmable Logic

Hardware designs typically exploit a variety of commodity logic devices and require small amounts of "glue logic" circuitry to implement design dependent adaptations between the main system components. A combination of economic and design constraints motivated the development of a flexible device whose initial logic operation is "uncommitted". These devices would be later programmed by the system designer to implement a particular piece of glue logic circuitry.

These first generation of programmable logic devices are historically referred to simply as Programmable Logic Devices(PLDs). Although the same term may be applied to all generations of programmable device, unless otherwise mentioned, the remainder of this thesis will use the historical interpretation of the term "PLD" and reserve the expression "programmable logic" to refer to the wider notion of programmable logic device. In the same way, further generations of programmable devices will be explicitly referred to using appropriate terms as adopted by the community.

PLD architectures consist of two main components: a logic-AND array which feeds the inputs of a logic-OR array. Permuting which of the two components are programmable gives a series of PLD sub-classes. For example, a fixed AND-array and programmable OR-array is equivalent to a programmable read-only memory (PROM). When both arrays are programmable, the device is conventionally referred to as a programmable logic array (PLA) and with a fixed OR-array, the device is programmable array logic (PAL). PLDs use their AND-OR arrays to

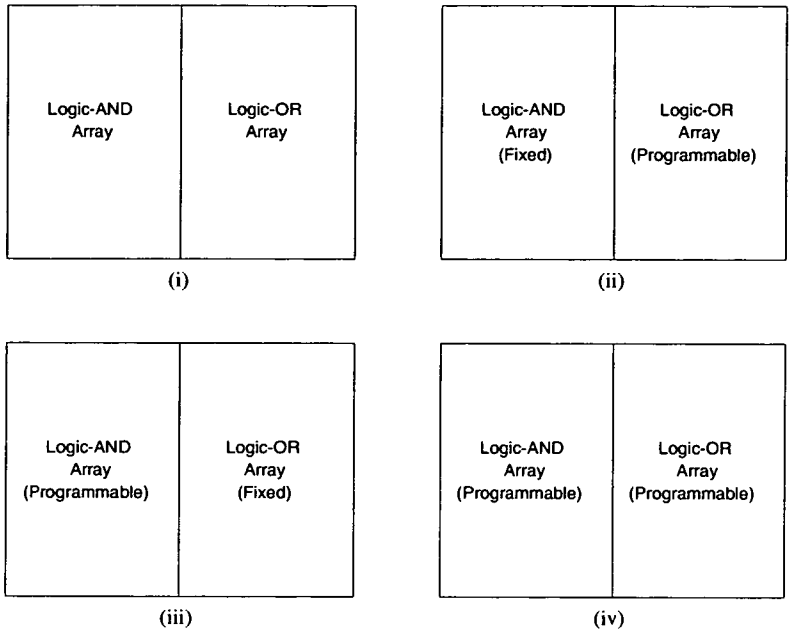

Figure 2.3: Sum-of-products PLD Architectures: (i) Basic PLD Organisation, (ii) Programmable Read-only Memory (PROM), (iii) Programmable Array Logic(PAL), and (iv) Programmable Logic Array (PLA)

implement simple boolean logic equations that can be expressed in a canonical, or "sum-of-products" form.

Early PLDs were typically one-time programmable devices that used an antifuse technology to implement their programmability: the device is programmed by effectively "blowing" fuses at strategic points in the architecture. Antifuse technologies are particularly appropriate for glue logic applications, which normally require very low pin-to-pin latencies. The act of blowing an antifuse creates an actual, physical conductive path in the underlying silicon substrate thereby eliminating the propagation delay incurred by the active circuitry used in other programming technologies. Newer generations of PLD [109] use advances in IC fabrication techniques to facilitate electrically-erasable devices which offer a degree more flexibility but without imposing much higher pin-to-pin latencies through the architecture.

The target applications of PLDs amount to small, well defined, simple combi-

natorial logic, e.g., address decoding or implementing small finite state machines (FSMs). This fits well with the low device density of PLDs.

The reasons for this are threefold and involve both architecture and market influences:

- The fundamental architecture of PLDs does not scale well: as the architectural parameters increase, so too does the size of the AND/OR array. Increasing the number of inputs, outputs or product terms, for example, has a non-linear effect on the silicon area and power consumption of the device. Some PLD architectures attempt to circumvent this by segmenting the AND-OR array into pages [51].

- In addition to not being scalable, implementing arbitrary logic in a sum-of-products form is not generally appropriate. PLDs, unlike the FPGAs discussed in the following section, are not register-rich devices. Only a few registers will be provided within a PLD and these are mainly used to latch inputs and outputs at the device periphery. Even advanced PLD architectures in production in 2000 [109] have very limited numbers of register components, making the implementation of complex, stateful calculations difficult.

- Finally, PLDs were efficient at implementing the simple glue-logic applications they were targeted at. Since the application domain itself was limited to such relatively small designs, the market demand for high density PLDs remained low. The most significant increase in the size of glue logic applications has come from implementing more complex FSMs such as DRAM controllers.

2.3 Field Programmable Gate Arrays (FPGAs)

Whilst PLDs were the dominant form of programmable logic from the mid-1980s to the early 1990s, three new application domains for programmable logic appeared: Application-specific IC(ASIC) replacement; Rapid system prototyping; and Dynamic Runtime Reconfiguration. Motivated by these new applications and fueled by the availablilty of VLSI fabrication techniques, a new generation of programmable logic device arose in the form of the FPGA.

Each of the main FPGA application domains is characterised below.

2.3.1 ASIC Replacement

ASIC replacement has become the dominant driving application for FPGA devices. ASICs present one primary advantage to the system designer: their application specific nature means their implementation is tailored to exact the maximum performance for a defined application. To their detriment, however, ASICs require considerable expertise to develop, and, since they are fabricated directly into silicon, they also require a significant economic investment. Such high development costs must be either amortised through large production runs to bring down the unit cost, or a high unit cost must be justified by becoming the dominant solution in a defined niche-market. The architecture of FPGA devices is much more suited to implementing a wider range of logic circuitry than their PLD predecessors. FPGAs are register-rich architectures and, as will be noted in sections below, some architectures contain additional cell logic that makes them particularly efficient for certain application classes. The programmability of FPGAs also allows them to be tailored to implement a high performance solution to a particular application whilst, at the same time, avoiding the costly fabrication cycle required for ASICs. This, combined with the relatively inexpensive unit-cost per FPGA device, forms a compelling economic reason for replacing ASICs with FPGA devices.

Although the FPGA implementation of a circuit will be physically less efficient than a direct implementation in silicon, designers found that many of their application-specific solutions could achieve adequate performance when implemented in FPGAs. Furthermore, FPGAs have shown strong growth in both speed and density, so that more and more ASIC applications have become tractable in an FPGA implementation. As fabrication technologies have advanced, the regularity of FPGA architectures has allowed FPGA manufacturers to make more aggressive use of increased silicon real-estate than the majority of other VLSI applications which generally do not possess such architectural regularity. This has, in turn, become the driving factor in FPGA growth.

2.3.2 Rapid System Prototyping

In the second FPGA application domain, Rapid System Prototyping, the reprogrammability of FPGAs is exploited to decrease the time between design iterations of a system being developed. In contrast to ASIC replacement, the aim of rapid system prototyping is not to replace a system with one or more FPGAs, but to use the reprogrammability of FPGAs to quickly obtain accurate quantification of design metrics of proposed system designs. A very high-level view of a traditional approach to complex logic system design typically requires repeated periods of design capture and design simulation that eventually lead to the development of intermediate system prototypes. The construction of physical prototypes, akin to the development of custom ASICs, is expensive and time consuming. At the same time, however, it is necessary to ensure that the final system design will meet design constraints that cannot be completely guaranteed through simulation.

A rapid system prototyping approach using FPGAs, however, typically involves a design capture phase followed directly by an implementation phase where the design is mapped to a particular FPGA prototyping environment. The design

is exercised within that environment by configuring one or more FPGAs with the mapped system design, then providing them with real-time stimuli, and recovering actual results. The advantages of this approach are that, firstly, long periods of simulation can be reduced or avoided completely as the design can often run at up to system speeds on the FPGA prototyping environment. Secondly, the data obtained from an actual execution in the prototyping environment will be more realistic and reliable than those gained through simulation (which is typically conservative and pessimistic, essentially erring on the side of caution). Finally, the number of physical prototypes that must be constructed during the system design can be reduced, although probably not eliminated altogether as the final system will be a custom implementation of the rapidly developed prototype.

2.3.3 Dynamic Reconfiguration

Whilst ASIC replacement has become the dominant commercial application for FPGAs, the final application domain, Dynamic Reconfiguration, has become a focal point in the FPGA research community. The main goal of dynamic, run-time reconfiguration is to exact some degree of higher performance by dynamically adapting the logic circuitry used within an application. This is done by implementing at least part of the application's logic circuitry on an FPGA then harnessing the reprogrammability of FPGAs to specialise or alter the logic circuitry as the computational demands of the application change. An important requirement placed on FPGA architectures that support dynamic reconfiguration is that they be 'in-system programmable'. By this we mean that altering the configuration of the FPGA does not require its removal from the application hardware environment and installation in special-purpose reprogramming devices (some forms of Electrically Erasable PROM technologies require this). Instead, the FPGA has embedded control circuitry and a defined programming interface available to other devices in the system through its device pins. Different styles

of dynamic reconfiguration can be related to the exact timescales on which the FPGA is reprogrammed and the particular performance gain being sought. The primary difference to be noted in this application domain, however, is that it attempts to exploit all the features of FPGAs: their ability to implement complex arbitrary logic and their highly reprogrammable nature. In ASIC replacement, the reprogrammability of FPGAs is useful, but definitely not essential. Rapid system prototyping benefits from re-programmability, but can be served, if at greater expense, by high density FPGA architectures which are one-time programmable. Further, even when rapid prototyping demands reprogrammability, the timescales involved are quite different to those of dynamic reconfiguration: prototyping timescales are upwards of hours and days, whilst dynamic, runtime reconfiguration timescales, at their coarsest measure, are downwards of minutes and seconds.

The topic of dynamic, runtime reconfiguration is central to this thesis and through the following sections and chapters will be explored in much more detail. The remainder of this chapter, in particular, will present variety of FPGA architectures and systems which are relevant to the dynamic reconfiguration. For a wider review of ASIC replacement and rapid system prototyping, the reader is referred to the wider literature within the FPGA community [2, 3]. Rapid system prototyping is also well served by a series of dedicated international workshops [1].

2.4 Basic FPGA Architecture

A basic FPGA architecture has three main components: a collection of programmable logic blocks; a programmable routing infrastructure; and a number programmable input-output blocks(IOBs). FPGA architectures commonly have a symmetric organization, with logic blocks laid out in a grid structure. The routing infrastructure is usually organised as channels that run horizontally and

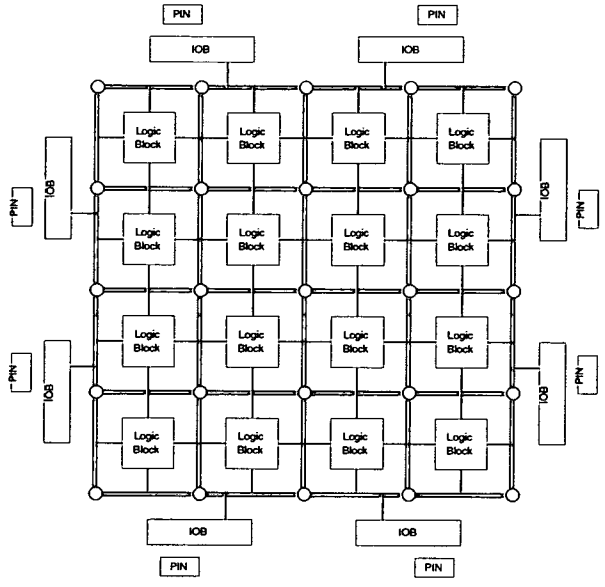

Figure 2.4: The basic architecture of an FPGA

vertically between the rows and columns of logic blocks; it is also common to augment these channels with direct nearest-neighbour routes between logic blocks. A series of IOBs are located around the periphery of the array with the primary purpose of allowing logic circuitry to interact with the FPGA's device pins. A generic FPGA architecture of this style is shown in Figure 2.4.

At this stage, the resemblance between the basic architecture of an FPGA and the cellular arrays discussed earlier is much clearer. The programmable logic blocks of the FPGA are equivalent to the cells of a cellular array, and the programmable routing infrastructure is equivalent to the cellular array's interconnect network. Indeed, FPGAs can be considered as VLSI implementations of evolved, highly flexible versions of the relatively simple cellular arrays that were being designed in the 1960s. For the remainder of this thesis, the term "cell" is adopted as a short hand for "programmable logic block" and aspects of the routing infrastructure of the FPGA may be referred to simply as routing.

Within this basic FPGA architecture, there is a large amount of scope for architectural diversity: the main computational elements of cells can be based on lookup tables(LUTs), multiplexors (MUXes), or combinations of basic logic gates;

the number of state elements in each logic block and the permutations it may form with the computational logic elements are variable; the layout of cells need not be a simple array, or even restricted to two dimensional geometries; the routing infrastructure may be segmented into channels of various lengths, form buses spanning the entire length of a device, or adopt a hierarchical structure; and, IOBs can be either simple interfaces to device pins, perform complex signal adaptations, or provide logic circuitry with access to the internal features of the array. Far from being a complete enumeration, this list is merely a characterisation of some of the potential design variations.

In the following sections, the architectural details of some important commercial and research FPGA architectures will be presented. As mentioned earlier, these are presented, primarily, to highlight the features of FPGA architectures that make them appropriate to dynamic, runtime reconfiguration. Additionally, however, the collected architectural details also testify to the diversity within the FPGA design space.

2.4.1 FPGA Programming Technologies

The particular programming technology underlying an FPGA architecture will govern, at the lowest levels, how effective the device is for implementing dynamically reconfigurable circuitry. At higher levels, the details of the programming technology are somewhat abstracted behind the programming interface of the device. This forms a central theme to the discussion in Chapter 3.

By and large, FPGAs use SRAM to retain their programming information: a physical layer of SRAM underlies the main architecture of the FPGA. The values that are loaded into that SRAM layer directly influence the operation of the logic blocks, routing lines, and IOBs in the conceptual layer above. Since the configuration store is basically a memory, it can be loaded and reloaded with different configurations when required. The term bitstream is used to refer to the

collection of data values that must be loaded into the configuration memory in order to realise a particular circuit on that FPGA. 'Configuration' is a slightly more nebulous term that is often used to mean bitstream, but can also refer to the current state of the entire device – an FPGA can be said to have a particular configuration after being loaded with at least one bitstream.

SRAM is not the exclusive programming technology used in FPGA devices, but it supports more flexible degrees of reprogrammability. Some architectures are antifuse programmable [25] whilst others [26] use non-volatile 'Flash' memory. Flash re-programmable devices are slower to reprogram but retain their configuration state even after the FPGA is powered down. High-speed, dynamic reconfiguration demands a fast and flexible FPGA architecture and, for this reason, the remainder of this thesis will focus on SRAM re-programmable FPGAs. Unless explicitly stated, the term FPGA will imply an SRAM based architecture.

2.4.2 Classes of Reprogrammability

The organisation of the three fundamental features of an FPGA architecture define how well that architecture will support static logic circuitry. It is also possible to classify instances of a particular architecture based on the exact degree of programmability they support. The presence or absence of two attributes of an FPGA's configuration memory form the basis of this classification:

- Firstly, when one part of the configuration SRAM is to be altered, does the entire SRAM have to be reprogrammed or can selective regions of the memory be altered independently of others.

- Secondly, must the entire device be taken offline when a new set of values are being loaded into the configuration SRAM, or can existing circuitry implemented on the device remain active whilst changes to the underlying configuration store are being made.

The exact terminology used within the FPGA community for each of these classes remained the subject of some debate when this thesis was written in 2000. However, we will define three main classes of programmability as follows:

- The programmability base class is characterised by devices that require their entire configuration memory to be reprogrammed and must be taken offline during the configuration process. In this thesis we shall refer to these devices as *dynamically reconfigurable.* This is the least flexible of the three programmability classes.

- The first extension to the base class adds a degree of flexibility by allowing circuitry to remain active whilst a new configuration is loaded. It is still necessary to load configurations for every cell, routing switch, and IOB in the architecture in each configuration cycle. We shall refer to these devices as *multiplanar.*

- The third programmability class is evident in devices where only the relevant parts of the configuration RAM need to be altered and this can be done whilst other circuitry in the array remains active. Device architectures of this type are referred to as *partially reconfigurable.*

Of the three classes, dynamically reconfigurable and partially reconfigurable are the two most common forms of FPGA. A number of multiplanar style devices exist [39, 31, 101, 89] and are often referred to as 'multicontext' or 'time-shared'.

2.5 Dynamically Reconfigurable FPGAs

2.5.1 Xilinx LCAs

The first commercially successful FPGA architecture was introduced by the semiconductor company Xilinx in 1985. Their architecture, termed a Logic Cell Array(LCA) and shown in Figure 2.5, was dynamically reconfigurable and has been

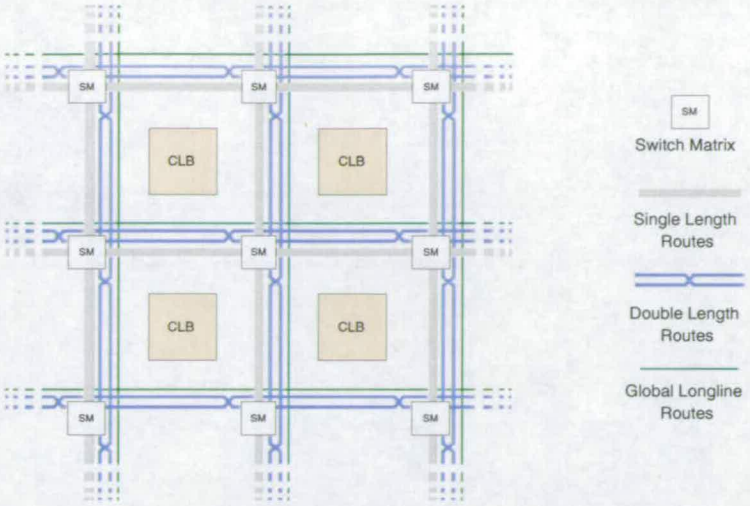

Figure 2.5: General Features of the XC4000 Cell Array

at the core of three generations of Xilinx LCAs. In this section, the XC4000 series is used to characterise the Xilinx LCA architecture.

Cells in the Xilinx architecture are called configurable logic blocks(CLBs) and are arranged in a simple two-dimensional grid. Figure 2.6 shows the XC4000 CLB structure. Xilinx CLBs are LUT based and, over the successive generations of LCA, the number of LUTs per CLB has increased. The CLB of the XC4000 has two independent 4-input lookup tables (f and g) capable of synthesising any function of their four inputs. A third LUT combines the outputs of the other two LUTs and one additional cell input, synthesising any function of the three inputs. Each cell also has two flip-flop state elements and, within later versions of the XC4000 series, LUTs not being used to synthesise combinatorial logic may instead be used as small, embedded data memories. One particularly important feature of the Xilinx CLB is the inclusion of dedicated logic to support fast carry propagation. Arithmetic intensive applications, such as many found within digital-signal processing (DSP), can exploit both of these features to gain performance.

The main programmable element in the routing infrastructure of the XC4000 is a programmable switch matrix situated between each CLB in the array. The exact configuration of each switch matrix defines how signals entering the matrix on one

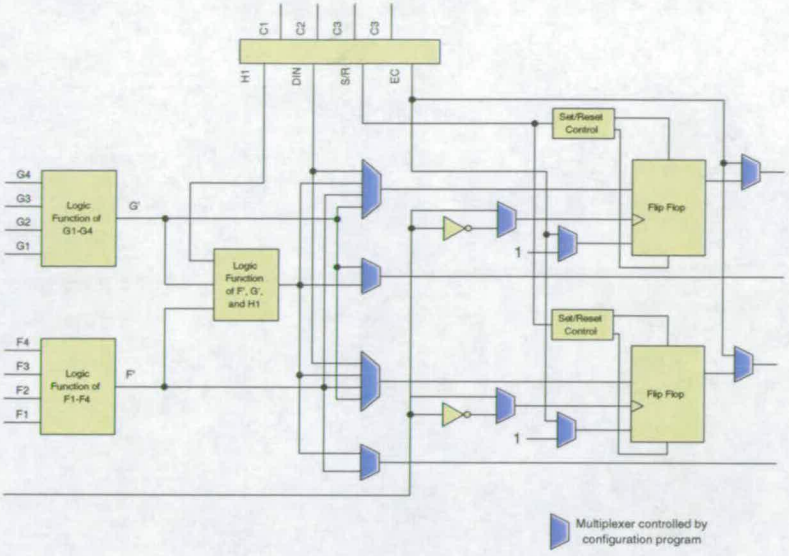

Figure 2.6: XC4000 Configurable Logic Block

side will be routed out on the other sides. For example, a signal entering from the top side of the matrix can be routed to one or more of the left, right, or bottom sides. Complex, irregular routes that exploit routing tracks of different lengths and orientations are constructed by configuring sequences of switch matrices. The XC4000 architecture has three main types of wired routing resource which are characterised by the relative length of their segments:

- single-length lines span exactly one CLB horizontally or vertically and intersect at the programmable switch matrices between each CLB;
- double-length lines span two CLBs in either horizontal or vertical directions and intersect at alternate switch matrices;
- finally, chip-length 'long-lines' span the entire length or width of the array and can be used by CLBs to connect with arbitrary CLBs in either the same column or row.

2.5.2 Altera Flex

The Flex architecture [27], produced by the semiconductor company Altera, is another example of commercially successful, dynamically reconfigurable FPGA. Although Flex devices contain the same three fundamental FPGA components, some aspects of the organisation of these components is radically different from the Xilinx LCA.

The first difference to note is that cells in the Flex architecture are grouped into clusters called Logic Array Blocks, or LABs. The exact number of cells contained within a cluster varies from device model to device model but, internally, cells contain LUTs, state elements, and dedicated logic to accelerate arithmetic operations. The cells in a cluster are interconnected through a routing resource local to the cluster itself. This local routing also serves as an access point to the main device routing, described below.

The second difference between the Flex architecture and the LCA architecture is that, in a Flex device, data memories are explicit components. Rather than converting unused LUTs into small data memories, as LCAs do, cell clusters are substituted for small blocks of embedded memory at various points in the array. These memories may be configured into various bitwidths by trading off the address space depth.

Finally, whilst the routing of a Xilinx array is based mainly on multi-length segments intersecting at switch boxes, the primary routing resource in the Flex architecture takes the form of long, unsegmented routing channels. These multi-bit wide channels run horizontally and vertically between the cell clusters and embedded memory blocks that connect to them. The unsegmented nature of this resource means that, at a physical level, signals are propagated faster as they do not incur delays as they pass through switch boxes.

In a very general sense, the Flex architecture rewards logic designs that map well onto cell clusters. If a design subcomponent may be mapped, in its entirety,

to a particular cell cluster, it can exploit the fast local routing within the cluster for its interconnect. Designs which do not partition well into clusters would ultimately consume more of the unsegmented tracks between clusters as signals internal to the subcomponent get mapped to 'global' wires. The knock-on effect from this is that placement and routing of the design becomes much less tenable as the routing infrastructure becomes congested.

2.6 Partially Reconfigurable FPGAs

The majority of commercial FPGA devices are dynamically reconfigurable devices. Of particular relevance to this thesis, though, there are some notable partially reconfigurable devices which have been instrumental in dynamic reconfiguration research.

2.6.1 The Xilinx XC6200

The Xilinx XC6200 [109] series has, arguably, had the most significant impact of any partially reconfigurable device on the field of runtime reconfiguration. The XC6200 series is an evolution of the Algotronix CAL [58, 5] architecture which, in turn, draws on the function synthesis approach used in the Cutpoint cellular arrays described earlier. Indeed, many of the architectural features of the XC6200 are quite different from the 'mainstream' dynamically reconfigurable architectures.

The cells of the XC6200, shown in Figure 2.7, are 'fine grained'. The granularity of cells is often used as a broad means of classifying different FPGA architectures: depending on the style of cells it contains, an FPGA is said to be either fine-grained or coarse-grained. The exact definition of these terms is nebulous and there is no real quantification of when an architecture stops being fine-grained and starts being coarse grained. Rather, the distinction is based on the relative complexity of the logic function that a cell is capable of synthe-

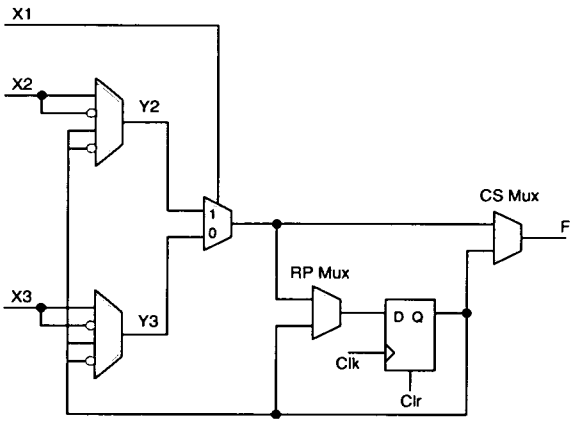

Figure 2.7: The XC6000 Function Unit

sising, with respect to other architectures of the same generation. In this way, the LCA and Flex architectures described above are examples of coarse grained architectures whilst the relatively simple XC6200 cells make it fine-grained.

The XC6200 cells operate in a very different manner to those of the Xilinx LCA. Each cell contains a set of configurable multiplexors and a state element. Combinatorial functions are synthesised by configuring the flow of input bits through the cell multiplexors in a particular manner. Sequential functions use the state element at the cell output and a feedback path connecting the output of the state element to the cell inputs. One notable omission in the cell architecture, however, is the lack of any dedicated carry propagation or cascade logic.

The geometric layout of the architecture and its routing infrastructure adhere, mainly, to the hierarchic organisation shown in Figure 2.8. Cells and nearest-neighbour connections between cells form the lowest level of the hierarchy. Above this, cells are grouped into 4×4 clusters. Dedicated switch multiplexors, placed at the periphery of each 4×4 cluster, to provide access to length-4 wires which interconnect adjacent 4×4 clusters. Similarly, at the next level, cells are grouped into 16x16 clusters with length-16 wires to interconnect them. Rather than having physically separate 16×16 switch multiplexors at the edge of a 16x16 cluster, additional switches for length-16 interconnects are provided in the 4×4 switch

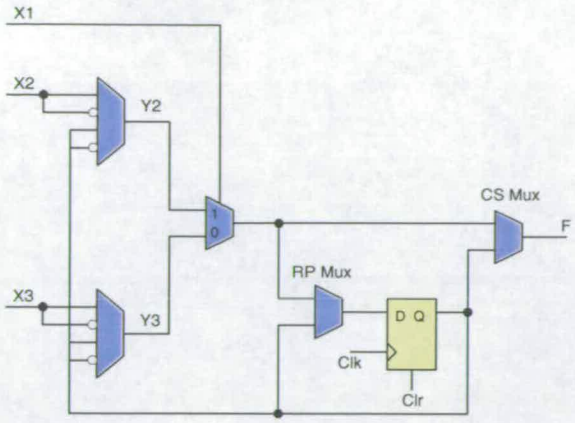

Figure 2.7: The XC6000 Function Unit

sising, with respect to other architectures of the same generation. In this way, the LCA and Flex architectures described above are examples of coarse grained architectures whilst the relatively simple XC6200 cells make it fine-grained.

The XC6200 cells operate in a very different manner to those of the Xilinx LCA. Each cell contains a set of configurable multiplexors and a state element. Combinatorial functions are synthesised by configuring the flow of input bits through the cell multiplexors in a particular manner. Sequential functions use the state element at the cell output and a feedback path connecting the output of the state element to the cell inputs. One notable omission in the cell architecture, however, is the lack of any dedicated carry propagation or cascade logic.

The geometric layout of the architecture and its routing infrastructure adhere, mainly, to the hierarchic organisation shown in Figure 2.8. Cells and nearest-neighbour connections between cells form the lowest level of the hierarchy. Above this, cells are grouped into 4×4 clusters. Dedicated switch multiplexors, placed at the periphery of each 4×4 cluster, to provide access to length-4 wires which interconnect adjacent 4×4 clusters. Similarly, at the next level, cells are grouped into 16x16 clusters with length-16 wires to interconnect them. Rather than having physically separate 16×16 switch multiplexors at the edge of a 16x16 cluster, additional switches for length-16 interconnects are provided in the 4×4 switch

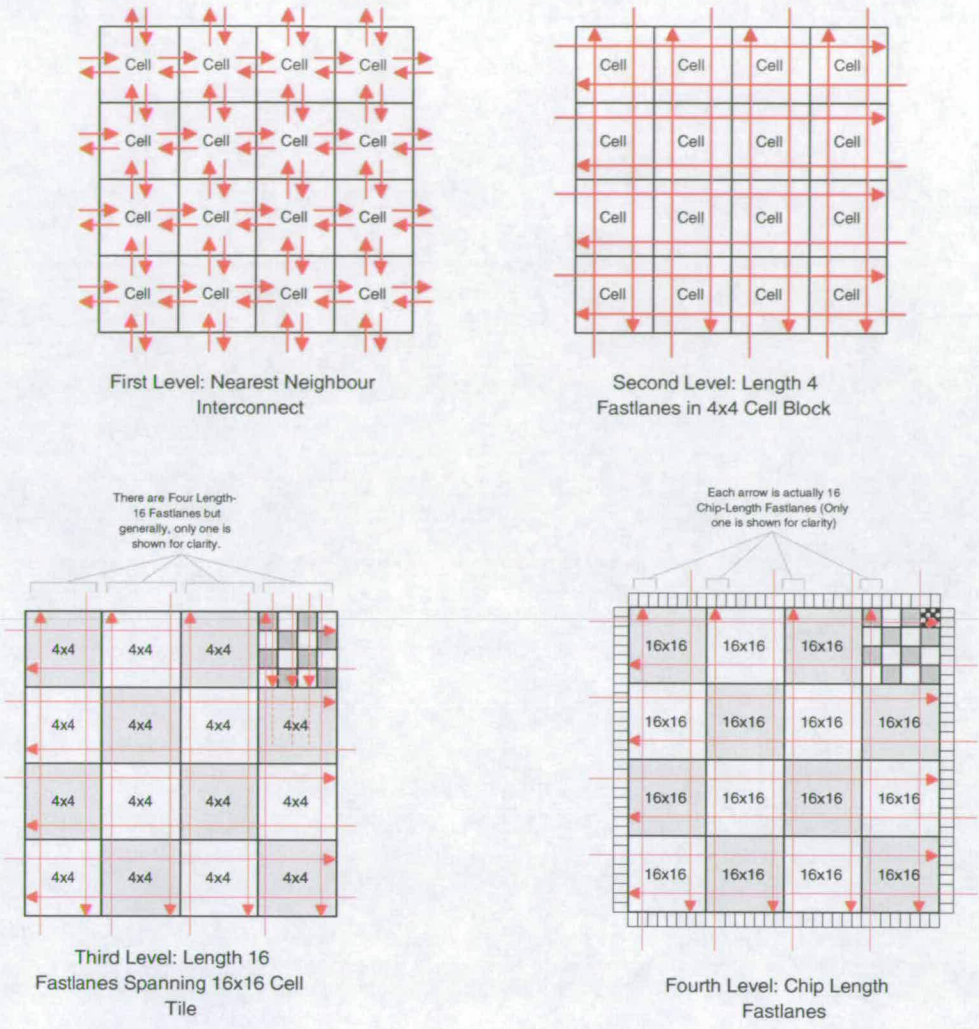

Figure 2.8: The XC6000 Routing Hierarchy

multiplexors that align with the boundary of a 16×16 cluster. The remainder of the device geometry is constructed from tiles of 16×16 clusters, whilst the top level of the routing hierarchy provides interconnects which span the entire width and height of the array, are also available through the switching multiplexors at the 16×16 cluster edges.

The hierarchical organisation of cells and routing is intended to provide a logarithmic scaling of signal delay as the distance between communicating cells increases. This is in contrast to most other architectures where the scaling of signal delay tends towards linear as the distance between cells increases. The

sources. Here, a device may be reconfigured from a remote station. The dangers of this style of system for architectures that are susceptible to signal contention is discussed in the literature on 'FPGA Viruses' [47].

One final noteworthy feature of the XC6200, but one that will not be considered in depth at this point in the thesis, is its programming interface. In stark contrast to the other architectures of its generation, the XC6200 has a very rich programming interface to the configuration SRAM of the device. The FastMap [21] interface is a microprocessor style interface that presents the configuration of the device to the outside world through a set of address, data, and control pins. Every part of the SRAM which controls the configuration of the device and the SRAM which contains the current logical values of the cell state elements is addressable. By simply reading and writing addresses through the FastMap interface, the device can be reconfigured. A fuller treatment of the features of FastMap interface, and its place in a continuum of interface styles is given in Chapter 3.

2.6.2 The Atmel AT6000

The Atmel AT6000 series [8] is an alternative example of a commercial, partially reconfigurable FPGA and also has a cell architecture, shown in Figure 2.9 unlike any of the FPGAs described above. Function synthesis in the cells of the AT6000 uses multiplexors to orchestrate the flow of input signals through a series of fixed logic gates and a state element. By using the multiplexors and feedback paths within the cell structure, the fixed logic elements can be organised in a number of different permutations. Unlike the use of multiplexors in the XC6200 architecture, which actually serve as computational elements, the multiplexors of the AT6000 are control elements whose selection of output is governed entirely by the configuration SRAM and cannot be directly influenced by the output of any other component in the cell. Like the XC6200, however, the Atmel 6000 cell contains

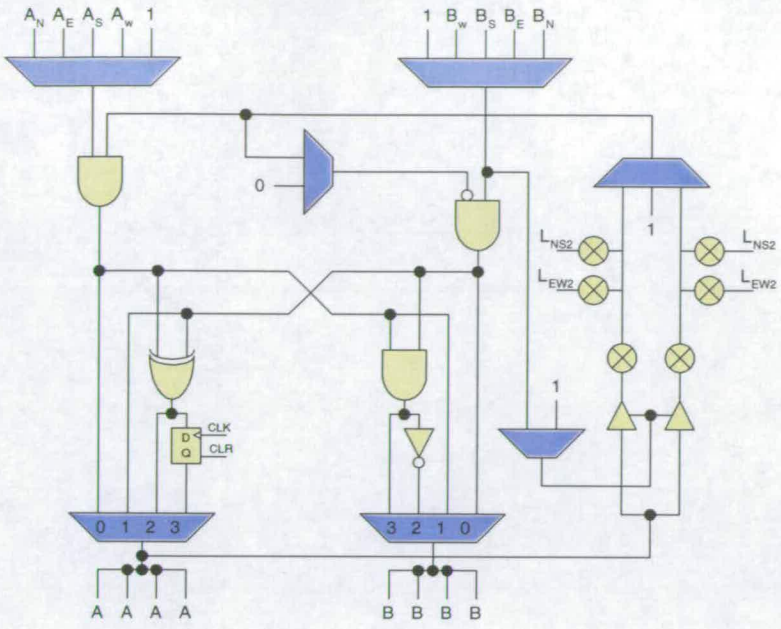

Figure 2.9: The AT6000 Cell Structure

no dedicated logic for arithmetic carry or cascade chains.

The geometric organisation of the Atmel array bears some similarities to that of the XC6200. Cells in the array are organised in a grid which is then partitioned into tiles of 8×8 cells by repeater units used in the routing infrastructure. The routing resources available are a combination of nearest neighbour, local bus, and express bus. Local and express buses form routing channels that run horizontally and vertically between each row and column of cells. Cells within a cluster can exploit their adjacent local buses within the cluster in much the same manner as length-4 routes serve the cells of a 4×4 cluster in the XC6200 architecture. Express buses are less segmented than local buses as they may only be driven by a local bus when they both intersect at a repeater. As a result, the express buses propagate signals more quickly across their length. Essentially, the rôle of a repeater unit is to provide endpoints for local and express bus segments, join adjacent local and express segments of the same orientation, and provide an intersection point for signals to traverse between local and express buses. This organisation is shown in Figure 2.10. Whilst all routing in the XC6200

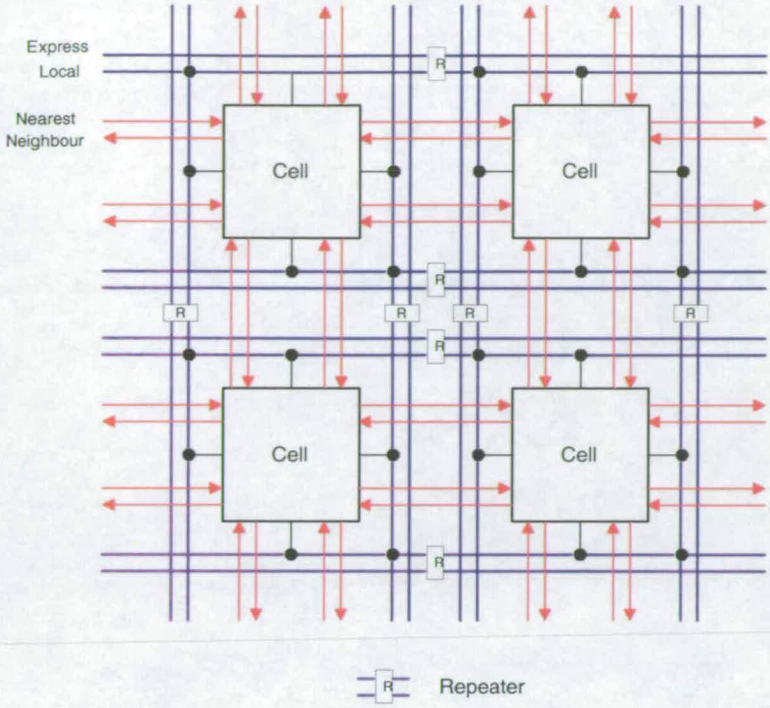

Figure 2.10: General Organisation of the AT6000 Array

architecture was uni-directional, however, long tristate buses can be constructed in the Atmel architecture.

Neither the XC6200 nor the AT6000 series remain in production, although a successor to the AT6000 was designed and released. The AT40K [9] is an evolution on the basic architecture of the earlier series and includes a set of performance enhancing features that make the AT40K particularly effective for implementing a class of DSP functions. Firstly, within the AT40K cell, LUTs have replaced discrete logic gates as the main computational element. As shown in Figure 2.11, two 8-input wide LUTs are combined with a single state element. In terms of granularity, the AT40K is of much coarser granularity than either of the earlier partially reconfigurable architectures.

Secondly, the device architecture is extended to include nearest neighbour routing in the four diagonal directions, making AT40K cells octagonal in shape. The inclusion of diagonal routing resources simplifies the construction of multi-

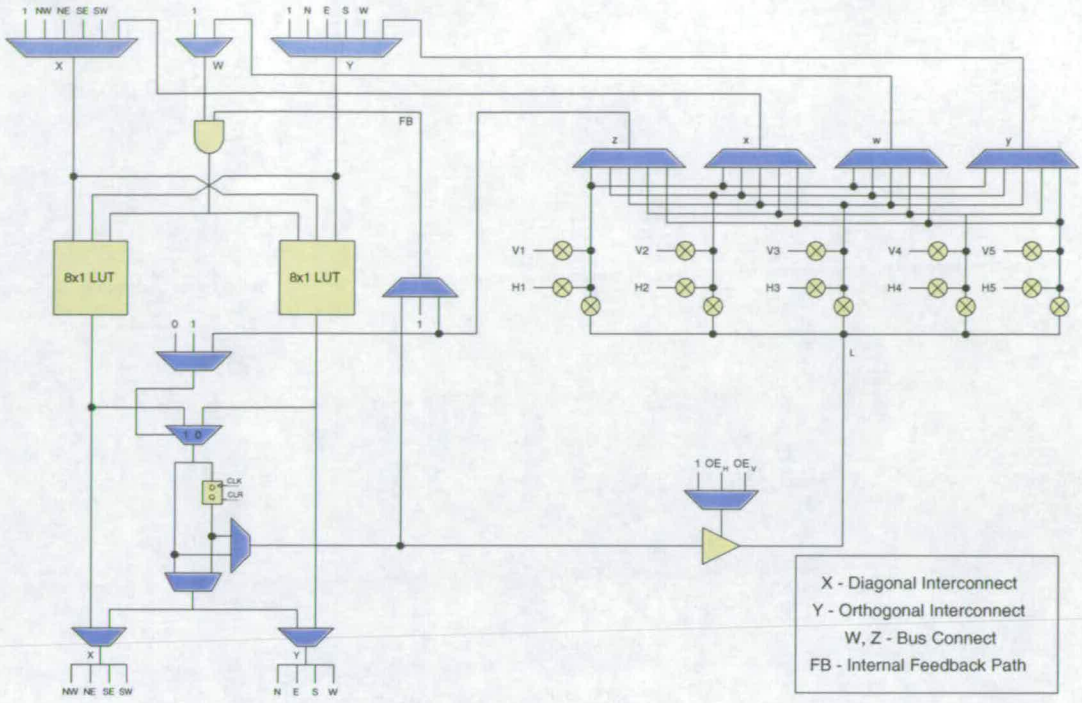

Figure 2.11: The AT40K Cell Structure

plier circuitry, which is used heavily in DSP applications. Overall, the geometry of cell layout within the AT40K keeps to the same clustered grid of cells, as used in the earlier AT6000s, but with fewer cells per cluster. Finally, small block memories are distributed throughout the array at the crossover points of the routing channels that run horizontally and vertically between the 4x4 cell clusters. The overall geometric layout of the architecture is shown in Figure 2.12.

2.7 New Generation FPGA architectures

As mentioned earlier, the current set of VLSI design practices do not scale to the integration levels being offered through advances in fabrication technology. As a result, VLSI designers are migrating from traditional VLSI design techniques to SLI design. The regularity of FPGA architectures has consistently positioned them to aggressively exploit increasing transistor counts. As FPGA architects begin to exploit the same fabrication technologies being used by SLI designers, a

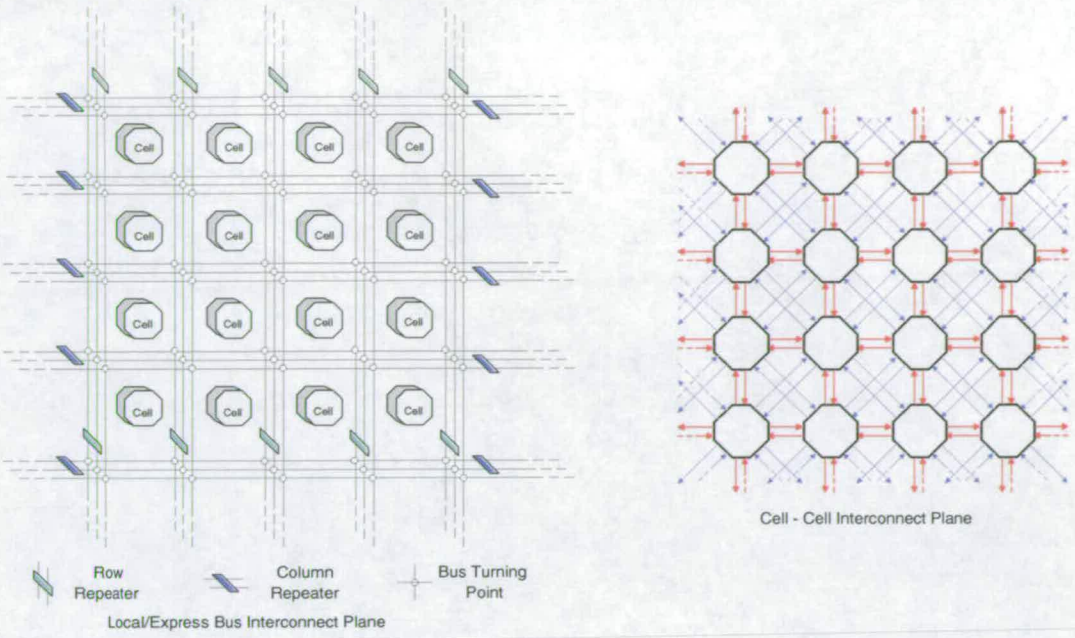

Figure 2.12: General organisation of AT40K Cell Array and Interconnect

new generation of FPGAs is being formed. At the moment, the main examples of such FPGAs are devices with architectures that scale to implement circuitry beyond one million equivalent gates. In the longer term, however, the important distinguishing features of the new generation architectures will be the facilities they include to counteract the physical effects of SLI fabrication processes. Furthermore, the usefulness of embedded memory blocks in successful first generation architectures has elevated embedded data RAM to now being one of the fundamental building blocks of a new generation FPGA. So far, the four fundamental components of a new generation architecture are: cells; routing resources; IOBs; and embedded memory blocks.

2.7.1 Virtex

The Xilinx Virtex is the first commercial example of a new generation FPGA and is similar to the architecture of a first-generation Xilinx series, the Xilinx XC5200 [109]. The high level organisation of the Virtex is shown in Figure 2.14 and its cell structure is shown in Figure 2.13. Like the earlier XC4000s, the Virtex array

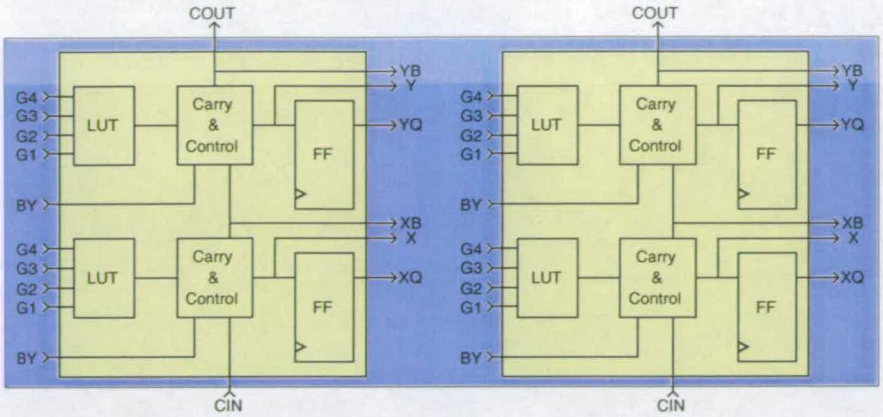

Figure 2.13: A Virtex 2-Slice CLB

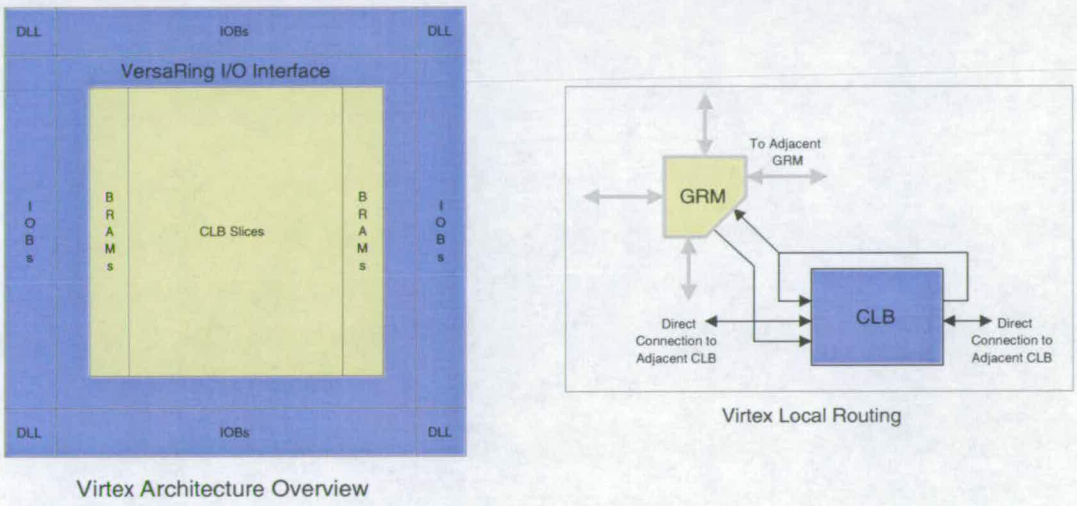

Figure 2.14: General Organisation of the Virtex Architecture

is a two dimensional grid of cells separated by horizontal and vertical interconnect channels.

A Virtex cell is built hierarchically from a basic collection of "logic cells" where each logic cell contains a four-input LUT, some dedicated carry logic, and a state element. From here, two logic cells are joined to form a single "slice". The slice joins the carry propagation logic of the two individual logic cells and, under certain circumstances, allows the outputs of the logic cell LUTs to be combined themselves and synthesise a logic function of five inputs. Two independent slices then form the contents of a single Virtex CLB.

The routing resources of the Virtex can be broadly split into three categories: a set of general routing resources that efficiently and flexibly interconnect the CLBs at various points in the array; a routing resource local to each CLB that serves to connect the individual slices and logic cells within that CLB, whilst also giving internal components of the CLB a wide access port to the general routing resources through a General Routing Matrix(or GRM); and, lastly, a smaller set of dedicated routing resources that provide a particular style of interconnect to a set of CLBs.

The isolation of the local CLB routing from the general resources through the GRM gives an important degree of mobility to subcircuits mapped into a particular logic cell or slice. A subcircuit can be remapped to a different part of the CLB whilst still retaining its connection to subcircuits in different CLBs. The changes to the circuit placement, when contained within the CLB, would not affect any components beyond that CLB's general routing matrix. Furthermore, the Virtex has a very rich routing infrastructure consisting of multiple, wide routing channels that interconnect the cellular resources of the array.

The local routing resources of a CLB perform three main functions: firstly, they provide an interconnection between the CLB LUTs, CLB state elements, and the GRM; secondly, they provide a feedback path so that the outputs of the CLB may drive the CLB inputs with a minimum of delay; and, finally, they eliminate the delay of the GRM when communicating to certain neighbour CLBs by directly connecting horizontally adjacent CLBs.

The general routing resources of the Virtex are intended to form high-speed paths, of different lengths, for signals that travel between CLBs. In total, there are three types of path that intersect at a GRM: single length paths interconnect adjacent GRMs in all four compass directions; hex-paths also reach out in all four compass directions and span six CLBs before intersecting with a GRM. Furthermore, the distribution of hex-lines is staggered along the width and height

of the array; and chip-length "long-lines" form unsegmented spanning the entire width and height of the array before intersecting with a GRM. The dedicated interconnect resources consist of the routes that join the carry propagation logic of each CLB to the propagation logic in the CLBs vertically adjacent, and the tristate-capable horizontal lines each CLB may directly drive.

Beyond its architectural organisation, the Virtex is notable for its partial reconfigurability. As will be shown in Chapter 3, the exact manner in which it is partially reconfigurable is less ambitious than the XC6200 and the Atmel devices. Nonetheless, the availability of partial reconfigurability in such a mainstream FPGA architecture is interesting as it provides an insight to the current commercial tradeoff point between the value of the feature and the cost of the silicon area required to implement it.

2.7.2 Apex

The Altera Apex series is the second example of a next-generation commercial FPGA. There are two main differences (beyond device density) between the Apex and the earlier Altera Flex architectures. Firstly, in addition to the embedded memory blocks and cell clusters of the Flex, an Apex device also includes embedded product-term style blocks. Secondly, the Apex includes an extra layer of routing hierarchy in the form of a horizontal channel that interconnects a group of cell clusters, embedded memories and product-term components. The overall device organisation is shown in Figure2.15 and shows how the additional layer of routing is used to form heterogeneous mega-clusters. Beyond these main differences, however, the Apex architecture is a scaling up of the earlier Flex series.

2.8 Device Architecture Research

The previous section has characterised a number of commercial FPGA architectures. Collectively, these architectures identify an architectural norm within

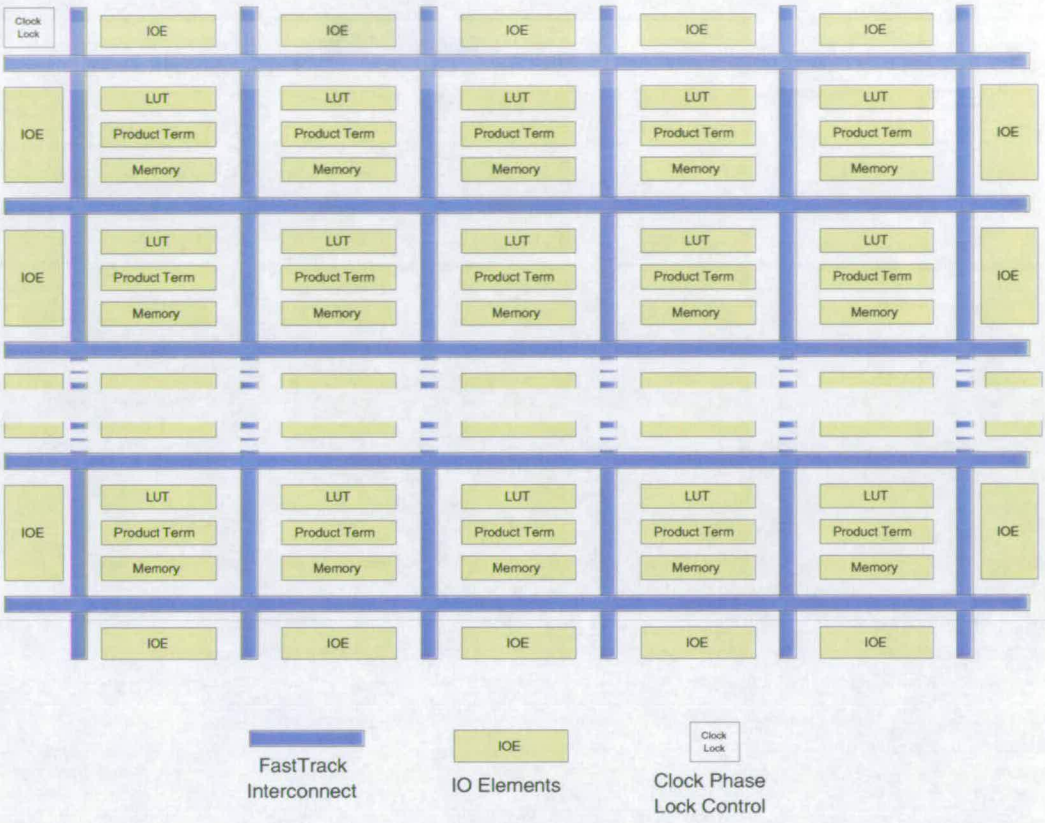

Figure 2.15: General Organisation of the Altera APEX

the FPGA design space which, in 2000, is characterised by LUT based, two dimensional FPGAs with hierarchical, segmented routing infrastructures. They are dynamically reconfigurable with coarse-grained core cells that can generally synthesise logic functions of up to approximately ten variables.

In addition to such mainstream, commercial architectures, there is also a significant amount of device architecture research. However, the intention in this thesis is not to give a comprehensive enumeration of such research architectures. Instead, we just note here that research into device architectures is exploring the FPGA design space beyond the architectural norm, on four broad fronts: array geometries and layouts [20, 37] ; routing and interconnect infrastructures [63, 33, 37, 83]; cell architectures and granularities [60, 38]; and device programming and configuration [20, 11, 83]. In the two subsections below we shall briefly characterise the architectural features of two research FPGAs. Whilst being ar-

chitecturally interesting in their own right, we consider them explicitly as their programming interfaces are of significant interest to the discussion in Chapter 3.

2.8.1 PipeRench

The PipeRench [20] architecture is designed specifically to support pipelined applications. In the FPGA architectures already discussed, the array's configurable resources were organised as a two dimensional grid of cells. In contrast, the configurable resources in PipeRench are organised on the granularity of 'stripes' where each stripe is roughly equivalent to a single pipeline stage. Figure 2.16 gives a general view of the PipeRench configurable fabric's structure. The contents and internal structure of a configurable stripe are considered in [90, 61]. We should note that the granularity of the device architecture is not chosen simply to ease the process of statically mapping a section of application logic circuitry to the device. Rather, stripes are primarily chosen to represent the atomic unit of reconfiguration. PipeRench attempts to support runtime reconfiguration more effectively by matching the device's atomic unit of reconfiguration to an appropriate level of abstraction in the architecture's target application class. As such, we can broadly classify PipeRench as a pipeline reconfigurable [90, 69] architecture.

2.8.2 Colt

The Colt architecture [11], like PipeRench, is designed to support runtime reconfiguration for a particular class of applications. The basic architecture of a Colt device is given in Figure 2.17. Colt implements Wormhole runtime reconfiguration as a means of supporting stream oriented computing and applications. In the architecture, streams of configuration and operand data enter the device through stream ports. Configuration control is distributed throughout the device, allowing streams to steer themselves through the array fabric, between the function units, over the crossbar interconnect, and out through a chosen stream port. Configuration data is stripped from the stream as it flows through the ar-

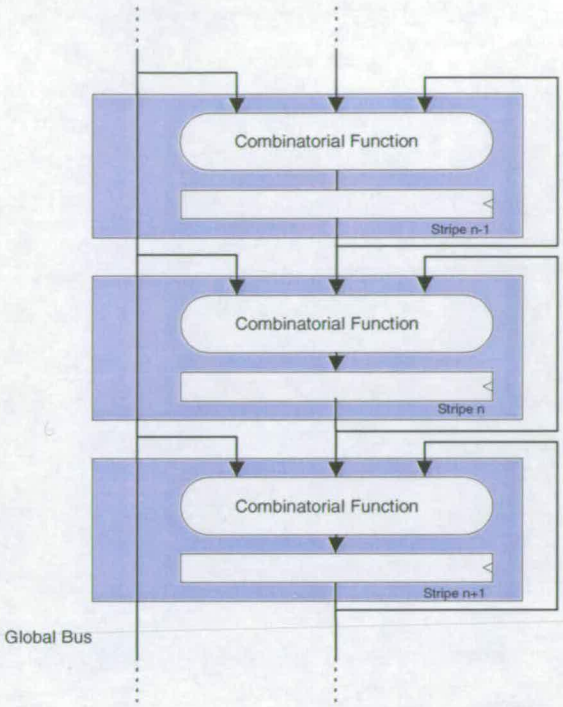

Figure 2.16: General Structure of the PipeRench Fabric and Stripe Functionality

chitecture's cell array. A similar approach is also taken in the abstract Plastic Cell Architecture [83].

2.9 Reconfigurable Computing Systems

On their own, the device architectures we have presented do not comprise a complete computing system and, typically, FPGAs are harnessed in larger system architectures with other forms of processing element. Whilst a comprehensive discussion of reconfigurable systems architectures is is beyond the scope of this thesis, we can identify two broad generations of reconfigurable computing system:

- first generation reconfigurable computing systems [42, 6, 7] are typically macro-architectures where the entire system is built from discrete devices interconnected on PCBs. Reconfigurable co-processor boards [107, 84, 80], where a FPGA subsystem is integrated within a standard PC-style host over the host's peripheral or system bus, fall within this class;

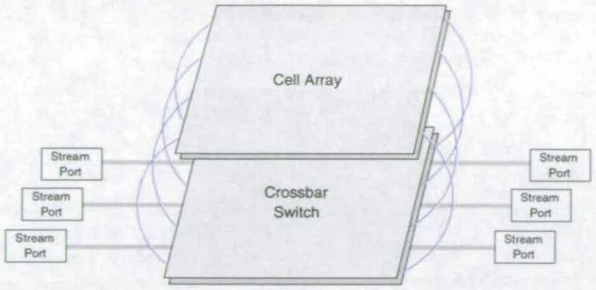

Figure 2.17: The Colt Architecture

- second generation reconfigurable computing systems, on the other hand, are integrated system-on-chip micro-architectures. The majority of second generation systems available in 2000 combine different forms of general purpose microprocessor with FPGA style reconfigurable logic on the same silicon die [102, 4, 39].

Macro-architecture style reconfigurable computing systems, and particularly reconfigurable co-processors, are often constrained by their low-bandwidth interfaces to the other system components. Indeed, the use of reconfigurable co-processors to accelerate general purpose processor systems is often thwarted by the constraints placed on the reconfigurable subsystem by the host's peripheral bus interconnect. However, we should also note that, even for system-on-chip microarchitectures, the inherent nature of a system architecture's style can still introduce bottlenecks between the system components.

2.10 Summary

In this chapter we explored the form and evolution of reconfigurable logic devices. We began with an exploration of early, historical devices and noted some of the key programmable logic systems that preceded the first generations of PLDs and FPGAs. From there, we considered the state of the art in programmable logic devices in 2000 through three successive generations of device architectures. This included a detailed exploration of the partially reconfigurable FPGA architec-

tures and technology that underpins the work described in the later chapters of this thesis. The chapter concluded with a description of some notable research device architectures whose form and philosophy are particularly relevant to the discussions in the forthcoming chapters.

Chapter 3

The Programmable Logic Interface

In this chapter we consider programmable logic devices beyond their physical architectures, and now within applications and dynamically reconfigurable computational systems. The discussion in this chapter has two main components:

- First we explore the concept of the programmable logic interface at different levels within dynamically reconfigurable systems. In particular, we use a short exploration of the structure and design of dynamically reconfigurable applications and systems to provide a context within which we can identify different abstractions of the programmable logic interface.

- In the second section we focus on the programmable logic device interface and explore different device interface styles in detail. In particular we consider the evolution of device interfaces from the relatively simple, serial interfaces used by early programmable logic devices, through to richer parallel interfaces, and on to streaming, protocol style device interfaces used in more advanced research device architectures.

3.1 Defining the Programmable Logic Interface

The general design flow for a mainstream FPGA architecture is very static in nature and is primarily oriented to the use of FPGAs for rapid system prototyping

or ASIC replacement. Despite this, there are many applications [107, 93, 59, 17] whose existence demonstrate the potential of runtime reconfiguration. Generally speaking, a dynamic, runtime reconfigurable application comprises three layers: high-level, application code; a runtime system supporting the mechanics of dynamic reconfiguration; and the low-level programmable logic subsystem. In this context we can identify three interfaces to the reconfigurable system's programmable logic, and these are discussed in the following sections.

3.1.1 The Programming Language Interface

The highest level of interface abstraction we consider is the programming language interface at the application level. No widely accepted design methodology and underlying theory that automates the systematic construction of runtime reconfigurable applications has emerged in 2000. The automatic synthesis of runtime reconfigurable applications from entirely behavioural problem descriptions is therefore difficult. Typically, it is the responsibility of the application designer to convey the partitioning and reconfigurable aspects of the application explicitly using the features of the programming language. This in itself is also difficult as the descriptive languages from either the software or the hardware communities do not express dynamic reconfiguration well. However, we can highlight two existing language mechanisms used to express dynamic reconfiguration: the dynamic instantiation of parameterised circuitry objects [79, 45, 10] in object-oriented languages such as C++ or Java; or partial evaluation in functional languages [99].

Whilst the complete design methodology for dynamic reconfiguration is still lacking, there are some proposed frameworks and partial methodologies [105, 46, 67], some approaches to the modeling and simulation of dynamically reconfigurable systems [75, 67, 106], and associated tool-sets [68, 72] which provide some degree of support to the dynamic reconfiguration application designer.

3.1.2 The Runtime System Interface

The main role of the dynamic runtime system is to provide resource management and communication facilities to higher level applications wishing to interact with the programmable logic subsystem. Ideally, such a runtime system provides an abstracted interface to the programmable logic subsystem, allowing it to be shared between multiple applications in much the same way that resources are shared in multitasking operating systems. However, for most runtime reconfiguration applications, the runtime management of programmable logic resources is typically implemented in an *ad hoc* manner and is highly application and system specific. More generalised runtime environments for dynamic reconfiguration have been suggested [19, 94, 104] for first generation reconfigurable co-processor systems.

3.1.3 The Device Interface

The lowest level interface is formed by the programmable logic's own physical, device interface. The challenges posed by the implementation of dynamic, runtime reconfiguration at this level are what we shall consider for the remainder of this thesis.

3.2 Programmable Logic Device Interfaces

In this section we characterise different styles of programmable logic interface at the device level. The most basic functionality in the programmable logic device interface is the mechanism for loading programming data into the device's configuration memory. In addition to this, however, the device interface may also facilitate access to the state of circuitry that is configured and active on the cell array. The form and semantics of a given device interface is influenced by its application domain and the device's architectural style. In the following subsections we explore the three main device interface styles and relate their

facilities to the architectures and application domains they target.

3.2.1 Bit-serial programming interfaces

The Xilinx LCAs discussed in the previous chapter are examples of FPGAs with a bit-serial programming interface. This is a consequence of the way that their configuration memories are structured: essentially, the entire configuration memory in an LCA is a single, long shift-register. Configuration data for the entire device is synchronously shifted into the configuration memory through a dedicated device pin. Whilst this interface style is suited to the ASIC replacement and rapid system prototyping application domains, it is much less suitable for the dynamic, runtime reconfigurable domain we are considering.

The main advantage of the serial style interface is its very low resource utilisation: very few device pins need to be dedicated to the programming interface and the silicon overhead of the configuration logic within the array architecture itself is small relative to the area of the array resources it controls. Both of these are relevant concerns in ASIC replacement and rapid system prototyping. Furthermore, the relative infrequency of reconfiguration in these domains means that the slow process of loading the programming data for the entire device bit-by-bit does not represent a significant overhead.

Support for interaction and interrogation of circuits configured in FPGAs with a serial interface is limited and, in most cases, the device interface only supports loading of programming bitstreams. In later generations of Xilinx LCAs, a configuration readback mechanism [28] is initiated through the FPGA's test-access port logic, the allocation of some device pins to output the configuration data and the status of the readback mechanism, and some additional circuitry configured on the array which links the device pins with the LCA's internal readback circuitry. When triggered, the readback mechanism serially shifts the current contents of the LCA's configuration memory out through a defined readback interface pin.

We should note that the mechanism does not explicitly read back the state of registers within the architecture. Rather, this information is distributed throughout the programming bitstream for the entire device. A suitable toolset [44, 64] can excise this data from the proprietary bitstream and allow the device state to be interrogated. Effecting changes to the state requires reconfiguring the entire device with a modified version of the read-back programming data.

3.2.2 Parallel, Random-access Interfaces

From the previous section we can see that the programming mechanism and interface of mainstream FPGAs is a significant drawback for implementations of dynamic, runtime reconfiguration. However, we can also identify partially reconfigurable architectures such as the Xilinx XC6200 which are inherently designed to support dynamic, run-time reconfiguration. From a physical perspective, the main difference between the device interfaces presented in this section and those of the previous section is the bit-parallel nature of the programming port. However, we should note that, as the density of LCA style architectures has increased, the bit-serial device interface becomes less convenient for loading the successively larger amounts of programming data required to configure the device. To combat this, it is common for LCA style architectures to also support bit-parallel device programming interfaces. For example, the Xilinx Spartan series of FPGAs is closely derived from the basic Xilinx LCA architecture but itself has a byte-wide 'Express'-mode programming interface [29]. This does facilitate an increase in the rate that programming data can be loaded into the configuration store of the device, but the fundamental programming mechanism within the device is still serial in nature.

In this section we will explore the device level interface of two partially reconfigurable architectures, the XC6200 and Xilinx Virtex. The Atmel series of FPGAs introduced in the previous chapter also provides support for runtime re-

configuration through their partial reconfigurability, but the proprietary nature of this underlying programming mechanism precludes them from this discussion. The interfaces of the XC6200 and Virtex are interesting here because the programming mechanisms they exploit are better suited to dynamic, runtime reconfiguration. In both cases, rather than simply evolving the serial configuration port to a bit-parallel version, the organisation of the architecture's configuration memory and its related programming mechanism are also evolved.

3.2.2.1 The XC6200 FastMap Interface

The FastMap [21] interface is a parallel, 'microprocessor' style device interface used in the Xilinx XC6200 that allows the the configuration memory of the device to be accessed as a RAM rather than a shift register. Furthermore, rather than just providing a mechanism for altering the configuration bits that govern the functionality of cells within the array, the interface also provides an integrated mechanism for directly reading and writing to the cell registers within the array. The FastMap is commonly described as a 'microprocessor' style interface because its basic semantics match those used on the memory bus of a microprocessor system.

Physically, the FastMap interface is formed by a set of address, data, and control signals. Table 3.1 lists the different FastMap interface signals[1] and describes their rôles. In this table we can see that only a subset of the FastMap signals are mapped to device pins on the array and a number of control signals are only available inside the cell array. A particularly interesting feature of the FastMap interface, and one which distinguishes the XC6200 from other FPGA architectures, is that the entire device programming interface can be accessed from within the array. This is considered further in the discussion contained in the next two chapters.

In addition to the explicit interface signals, the functionality of the FastMap

[1] signal names with a preceding / are active-low

Signal	Rôle
address	The address lines identify which word, nibble, or byte within the XC6200's configuration memory will be read or written. The exact width of the address bus is typically either 16 or 18 bits, depending on the exact model of XC6200 being used.
data	Configuration and state data being read and written to the array's configuration store arrive over the bi-directional data lines. There are 32 physical data lines, although the logical width of this port is, itself, configurable. The interface can be programmed to respond as an 8-bit, 16-bit, or 32-bit wide data port. This has a corresponding effect on how many bits of the address port are treated as significant.
/cs, rd/wr, and /oe	These signals have analogous rôles to their standard memory interface counterparts. /cs is a chip-select, rd/wr is a mode select indicating whether the current transaction is a read or a write to the configuration memory, and /oe is an output-enable signal controlling the driving state of the physical device pins.
/reset	An input signal that, when asserted, triggers a clearing of the device's entire configuration memory. This also places the FastMap interface circuitry in its default state, in which the XC6200 responds as if it were a basic SRAM.
GClk	A global clock signal to which all transactions over the FastMap interface are synchronised.
configOK	This is an internal signal asserted by the FastMap control logic when the FastMap ID register has been loaded with an appropriate value.
/RdEn and /WrEn	These are also both internal signals accessible from the East IOB serving each row of cells in the array. The signals are strobed when the FastMap interface is reading or writing directly to the registers within cells and can be exploited by user designs to detect when the registers in a given row are being accessed.
regword	Like /RdEn and /WrEn, regword is strobed when the FastMap interface is accessing cell state within the array. In particular, the signal can be used to detect when the FastMap interface is being used to access the state of cells in a particular *column* of the array.

Table 3.1: FastMap Interface Signals and their Rôles

interface is also influenced by a series of device configuration registers. In total, there are five sets of device configuration registers, all of which are also accessed via the FastMap's address, data, and control signals. The device configuration registers respond as distinct locations that are addressable within the XC6200's configuration store. To fully explain the influence the device registers have, we must first expand on the way that the FastMap interface is used to load configuration data and access cell state.

As well as presenting the configuration store as a RAM, the FastMap also contains additional logic to support fast circuit configuration and cell state accesses. These are the features that are primarily influenced by the device configuration registers. In its most basic form, using the FastMap interface to load circuit configuration data involves writing data words to the appropriate addresses within the configuration RAM. Address decoders within the FastMap interface circuitry route the programming data word to the correct region of the configuration store. Essentially, the bytes within the configuration store that govern each cell, IOB, routing switch, and device pad have defined addresses.

The first enhancement to the basic memory interface is the placement of a 32-bit mask register between the data port and the FastMap control circuitry. The register is used to identify which bits of the data bus are significant during a read or write to the XC6200's configuration RAM. During a FastMap write, if a bit is set at the nth position in the mask register, the corresponding nth bit in the word at the target address will not be affected. Similarly, during a read from the configuration RAM, if the nth mask register bit is set then the nth bit of the word asserted on the data port will be a logic zero rather than the contents of the configuration memory. When writing, the mask register can be used to preserve parts of the existing configuration word and, when reading, the mask register can be used to mask out irrelevant parts of the configuration word.

The second enhancement to the memory interface is the introduction of a

'wildcard' unit placed between the FastMap address port and its address decoding logic. The main function of the wildcard unit is to allow a write to a single address to actually write the same data to multiple addresses in the configuration RAM. Two device control registers are used in this process: the first wildcard register influences the row address decoder whilst the other influences the column address decoder. Any bit that is set in either register is treated as a 'don't care' bit by the associated address decoder. Consider, for example, if the column wildcard register is set to 00000001_2 and the column address before wildcarding[2] is 00101000_2. The column address decoder treats the lowest bit position as a 'don't care' bit and the column addresses that are actually activated during the write are 00101000_2 and 0101001_2. The primary motivation for supporting wildcarding is that circuits that are configured onto the array often have a regular, repeated structure. This regularity is reflected in the data that is written to the FPGA's configuration store, so the wildcarding mechanism is an effective means of broadcasting the repeated data simultaneously to the relevant parts of the configuration store. An alternative wildcarding mechanism, where the wildcard control registers are placed at the output of the address decoders, has also been proposed [55].

The mask and wildcard registers are primarily used for controlling FastMap writes to regions of the configuration RAM that govern the functionality of cells, IOBS, etc. We should note that, although the mask unit does not affect FastMap reads or writes to cell state, the wildcard unit does. FastMap state accesses occur on a column by column basis. That is, it is possible to read or write the state of the cells in a single column in each state access. However, since the column height exceeds the bit-width of the FastMap data port in every XC6200 family member, a mapping mechanism identifies which cells within a given column are targeted by the state access.

[2]the exact number of bits in the column address is dependent on the geometric size of the cell array. We are assuming that the column address in this example is for a XC6216 which has an 8-bit column address.

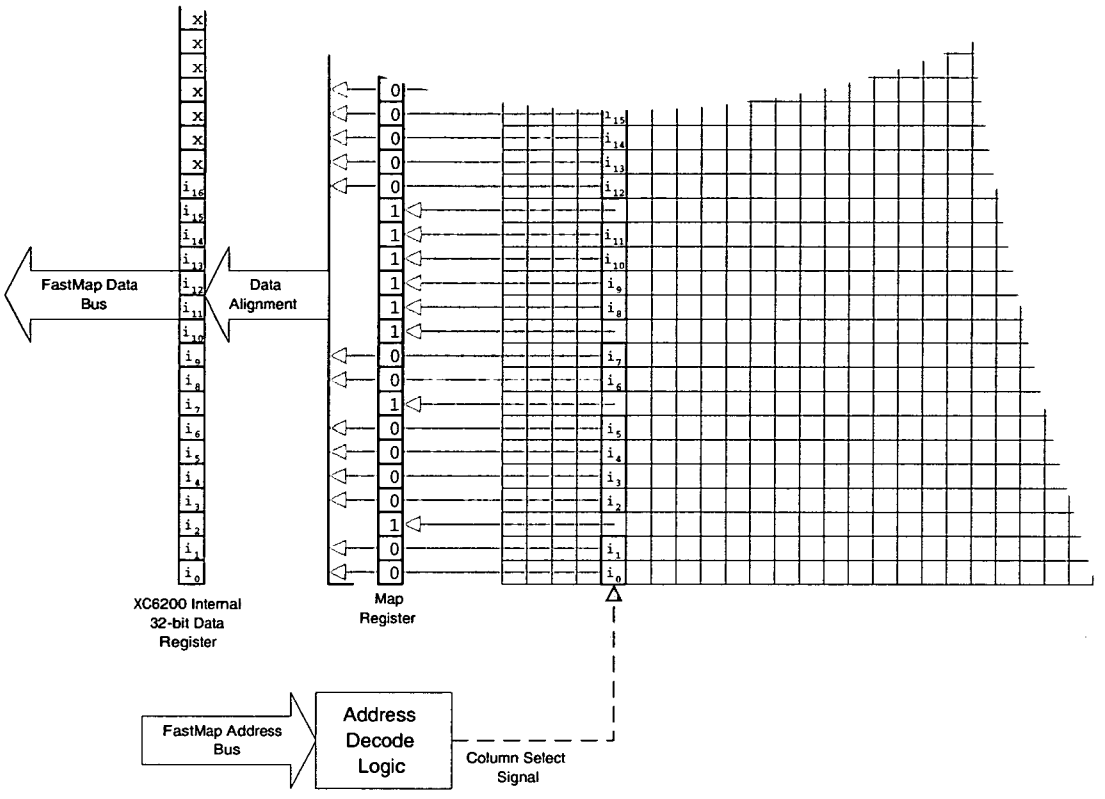

Figure 3.1: FastMap access to cell state using the XC6200 Map Register

Figure 3.1 shows the mapping mechanism being used to read the value contained in a 17-bit register placed in disjoint cells within a single column of the cell array. The mapping mechanism is controlled through a map register whose bitwidth matches the geometric height of the array's column of cells. Each bit position in the map register corresponds to a cell row and a logic zero in the nth bit position indicates that the cell in the nth row of the selected column is being accessed. Fundamentally this is a masking operation, similar to the masking facilities described earlier. However, in addition to selecting which cells are affected by the state access, the mechanism derives its name from the mapping operation it performs on the selected column bits. During a read, the selected bits from each cell in the column are packed onto the data port, starting with the row bit selected by the least significant bit in the map register. We can see this in Figure 3.1 where, even though the bits of the register are in disjoint cells of the column,

they are repacked in order, onto consecutive bits of the FastMap data port. The inverse situation applies during FastMap state writes. Here, the asserted bits of the map register define how the consecutive bits of the data port are distributed to the selected cells within the column.

The above discussion has introduced three of the five device configuration register types: the mask register, wildcard registers, and the map registers[3]. The "Device Configuration Register" and the "Device Identification Register" are used to influence general features of the device interface and do not directly facilitate fast circuit configuration or state access. Table 3.1 describes the relationship between the FastMap's `configOK` signal and the device identification register and we shall see that, the device identification register does have a significant rôle during the discussion in Chapter 5.

3.2.2.2 The Virtex SelectMap Interface

In 2000, the XC6200's FastMap interface remained the most flexible implementation of a programmable logic device interface. However, we can attribute a cost to such flexibility. Implementing the random access nature of the FastMap interface, in particular, incurs a significant silicon area cost. In this section we will briefly consider SelectMap interface of the Virtex architecture. In commercial terms, the Virtex series is considerably more successful than the XC6200 and, although the Virtex's SelectMap interface is substantially less flexible than the XC6200's FastMap, its form is indicative of a cost-flexibility tradeoff that is acceptable to mainstream FPGA vendors.

The Virtex architecture, like the XC6200, is partially reconfigurable and, physically, has a bit-parallel external programming interface. In contrast to both the FastMap and the earlier serial interfaces, the configuration memory of the Virtex is neither randomly accessible nor is it organised as a single, long shift register.

[3]Since the height of the map register matches the cell array height, it also exceeds the maximum bit-width of the data port. In practice it is therefore segmented and considered as multiple consecutive map registers.

The Virtex array is instead reconfigurable on a column by column basis and, whilst the underlying configuration store for each column is organised as a single shift register, the SelectMap interface allows columns to be reconfigured independently of each other. Unlike the FastMap interface, the SelectMap does not have inherent support for accessing cell state and embedded block-RAM state. Instead, the array uses the same readback mechanism employed by the earlier serial-style device programming interfaces.

3.2.3 Streaming, Packet-style Interfaces

In the previous sections we considered two dominant styles of programmable logic device interface. In this section we consider the programmable logic interfaces of the PipeRench and Colt device architectures introduced in Chapter 2. The interfaces of these architectures are significant as they represent a third style of programmable logic interface which supports runtime reconfiguration in a manner which is specific to the architecture's target application class. In particular, the interfaces of both architectures have a packet oriented approach to configuration and communication.

3.2.3.1 PCI-PipeRench

Figure 3.2 shows the basic structure of PCI-PipeRench [61], a prototype implementation of the PipeRench fabric. Physically, PCI-PipeRench interfaces to the host system through two 32-bit communication ports. Stripe configurations and application data are transmitted to the device as a formatted packet stream and arrive in the device through the 32-bit physical input port. The integrated, on-chip input controller decodes the stream into its constituent packets and takes action depending on the addressing contained in the packet's control section. The packet data is either directed to the dedicated configuration controller, forwarded to the striped fabric as application data, or immediately directed on to the output controller. The main task of the output controller is to format data received from

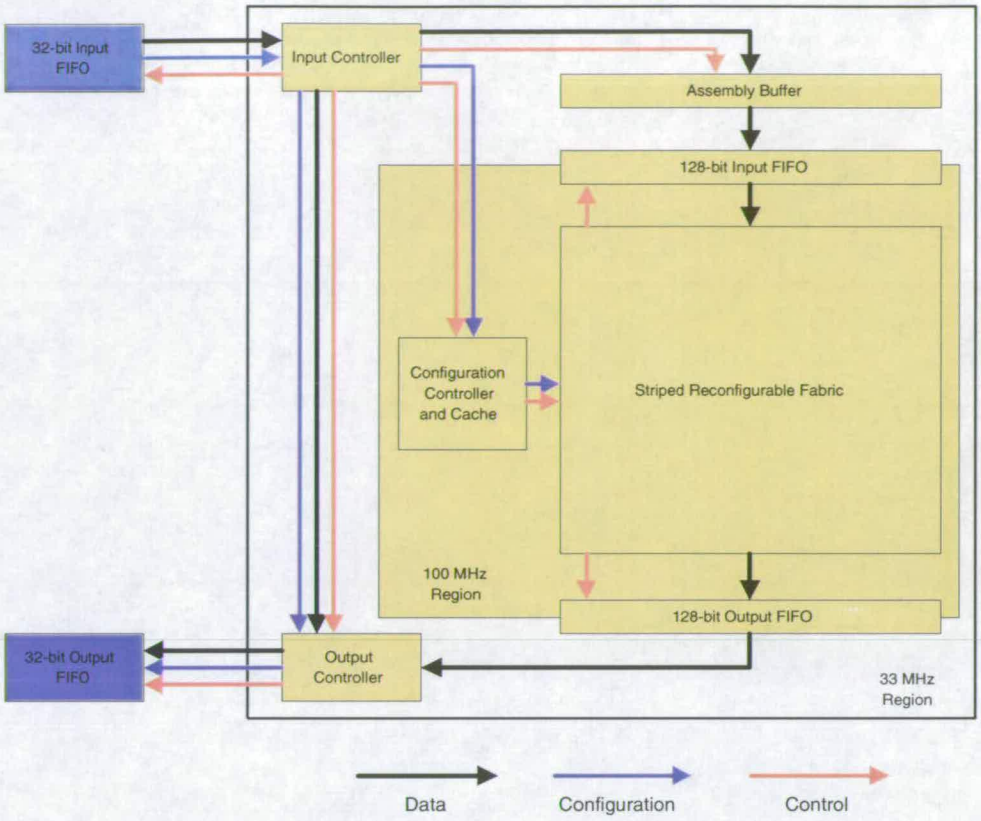

Figure 3.2: Structure of the PCI-PipeRench

the striped fabric into outgoing data packets. Additionally, the output controller re-sends packets that the input controller determines are not addressed to the current device.

The general format of a PCI-PipeRench stream packet is shown in Figure 3.3 and comprises three sections: a header, a marker, and payload contents. Both the header and marker are one word in size, whilst the payload may be dynamically sized. Generally, the header identifies how the packet should be processed whilst the marker is used to describe the contents of the packet payload. For example, the header word contains a simple form of addressing in the form of a chip-ID used by the input controller to determine whether it should consume the packet. The marker, on the other hand, generally specifies the amount of data within the packet payload. The input controller would then use this information to route the next n words to the appropriate part of the device.

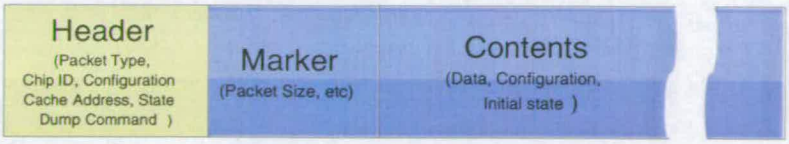

Figure 3.3: Typical Packet Format for the PipeRench Architecture

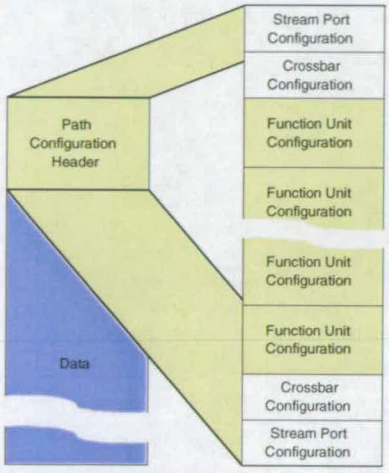

Figure 3.4: General Format of a Colt Stream

3.2.3.2 Colt

The general format of a Colt stream is shown in Figure 3.4. Colt takes a decentralised approach to configuration management and this is reflected in its stream structure. Whilst PCI-PipeRench uses explicit control headers to delimit the type and content of the incoming packets, the Colt stream has a path configuration header which is then followed by the application datastream. In Figure 3.4, we see that the path configuration header is built from packets of configuration data arranged in the order that they are consumed by each part of the distributed configuration control. The path configuration header allows the stream to guide itself through the array. An appropriate configuration packet is stripped from the path configuration header as the stream winds its way through the architecture. Specifically, each configuration packet contains all the data necessary to configure the device resource at the head of the path.

3.2.4 Adaptive Packet-style Device Interfaces

The PCI-PipeRench and Colt interfaces have one main limitation: although both device interfaces benefit from being tailored to a particular application class, the semantics of the packet interface is statically defined in the hardware implementations of, for PCI-PipeRench, the input and output controllers or, for Colt, the distributed configuration control logic. The remainder of this thesis considers a more flexible approach to packet style device interfaces. In particular, our aim is to develop a style of device interface where we can dynamically alter the semantics of the interface protocol to better support different application classes, or even specific applications.

3.3 Summary

In this chapter we considered the nature of the programmable logic interface at different levels of abstraction within dynamically reconfigurable systems. Focusing on the programmable logic interface at the device level, the chapter then considered the evolution of programmable logic device interfaces from their fairly simple serial origins, through to parallel interfaces such as the FastMap and SelectMap. We then considered newer forms of packet based, protocol oriented device interfaces used in the PipeRench and Colt architectures and, from there, approached the concept of a flexible, adaptive packet-style device interface. In the following chapters of this thesis, we will consider the implementation of such a flexible packet-style interface using the rich features of the XC6200 architecture to support the implementation.

Chapter 4

An Abstract Architecture for Virtual Circuitry

The previous chapter explored the concept and form of programmable logic interfaces which support dynamic, runtime reconfiguration. In this chapter we will describe the features of an abstract architecture which supports dynamic reconfiguration applications realised as virtual circuitry systems. The discussion has three main themes:

- First, we define the concept of virtual circuitry and discuss the two main models of virtual circuitry.
- Second, we introduce an abstract architecture which supports the existing models of virtual circuitry and broaden the discussion briefly to consider that abstract architecture within the class of Transport-Triggered Architectures (TTAs).
- Finally, we describe a third model of virtual circuitry using features of the abstract architecture.

4.1 Virtual Circuitry

Virtual Circuitry is a metaphor applied to runtime reconfiguration, typically in the context of FPGA based co-processor systems. Here, we wish to present the illusion of having a much larger programmable logic resource for the higher level

application software that wishes to exploit custom circuitry to gain increased performance. Specifically, VC encapsulates the act of dynamic, runtime reconfiguration as an analogue of virtual memory. Instead of swapping virtual memory regions in and out of physical memory, though, we dynamically swap circuitry on and off of the programmable logic subsystem.

In a virtual memory system, the granularity of virtualisation is balanced against the cost associated with each act of swapping to preserve an adequate level of overall performance. The same is true in VC where the cost of instantiating and removing logic circuits on the underlying FPGA must be balanced with its impact on the overall system performance [70].

There are a number of synonyms for virtual circuitry[1]; in the literature it is commonly referred to as virtual hardware [12, 98, 104], cache-logic [8] or logic caching [71]. From this, we can see that the rudimentary notions of VC have been present within in the FPGA community for some time. However, the advent of partially reconfigurable device architectures represents the watershed point beyond which VC became, practically, much more tractable. One of the most significant contributions [12] at that point was the introduction of two fundamental models of VC. A brief description of both models, set in the context of an FPGA with at least the dynamic reconfiguration facilities of the XC6200, is given in the following section.

4.2 Models of Virtual Circuitry

4.2.1 The requirements to support Virtual Circuits

Although the section above highlights the introduction of partially reconfigurable FPGAs as a watershed point in the tractability of VC, before we consider either of the two VC models it is useful to explore just what the requirements to support virtual circuits are. An important point to note here is that the specific

[1]We adopt the term virtual circuitry in this thesis for the reasons presented in [16].

quantifications of the requirements to support VC are heavily dependent on the particular VC application. Throughout this discussion, however, we should recall that the two fundamental operations that we must support in the VC environment are the configuration of a circuit onto the reconfigurable resource and the subsequent interaction with that circuit to process data.

The first requirement to support virtual circuits is that the target platform is a partially reconfigurable FPGA architecture. This is for two primary reasons: first, partially reconfigurable architectures allow us to support multiple independent virtual circuits on the same platform; and, second, partially reconfigurable architectures support SLU reconfigurations in the timescale of a few microseconds at the coarsest granularity. Early VC applications were targeted at dynamically reconfigurable FPGA architectures and have reconfiguration timescales on the order of milliseconds. Whilst this matched the reconfiguration timescales supported by that generation of architectures, it is less appropriate in contemporary VC applications. For such applications, we want to support reconfiguration typically in the order of hundreds of nanoseconds to a few microseconds. One specific example of this style of application would be within network routing switches. Here, high speed datastreams traveling over the network backbone require rapid reconfigurations to switching circuitry that is implemented within the reconfigurable array.

A second functional requirement relates to our ability to interact with the virtual circuits. Our general aim here is to be able to interact with the circuits at close to their core speeds, as much as possible. Early generation virtual circuit applications on architectures such as the Xilinx XC4000 series typically operate with clock speeds in the order of tens of megahertz. For the contemporary applications when this thesis was written in 2000, circuitry can operate at clock speeds of the order of 100MHz. Our functional circuit interfacing requirement, therefore, is that we should be able to support interactions with a circuit at up to its core

clock speed. Taking this further, the actual data throughput of a virtual circuit can be large because of the potential to support very wide datapaths. For high performance applications, it is feasible to support wide datapaths, on the order of hundreds of bits, within the target FPGA architecture. Overall, this means that our interfacing strategy for streaming data to and from virtual circuits must be powerful enough to supply such wide datapaths at core clock speeds.

Size is an important consideration for VC applications: we must ensure that enough reconfigurable resource is available to host the virtual circuits for the given application. This is analogous to balancing the ratio of physical to virtual memory in a virtual memory system to avoid unnecessary swapping and paging. Here we require that there is a balance between the amount of array resource dedicated to supporting virtual circuits, versus that allocated to the circuits themselves. A further concern here is that the scaling of the FPGA area itself should require at worst a linear scaling in the resources required to support the virtual circuits.

Again, the exact quantification of resource that is required is highly application dependent. However, we can generalize that the array resources required for a typical virtual circuit are on the order of hundreds of gates for the simplest circuits to tens of thousands of gates for complex circuits. In the context of contemporary platforms, we would wish to host such applications on platforms with a minimum of tens of thousands and, more likely, hundreds of thousands of gates. Further to this we would then restrict the reconfigurable resources dedicated solely to virtual circuitry management to at most 10% of the available reconfigurable resource. In 2000, this seems an appropriate figure to choose: VC applications targeted at the device architectures available in 2000 do not typically attempt to instantiate such large numbers of virtual circuits that a 10investment of array resource is unreasonable. Furthermore, in this context we are more likely to be challenged with constraints on reconfiguration bandwidth for each of the circuits, and physical constraints such as the available IO pins within the resource, before

we exceed constraints on the available reconfigurable resource itself.

We have noted in earlier Chapters that, towards the end of 2000, device densities were rapidly approaching millions of gates. When this class of device becomes the target for VC applications, we may conceivably see a reduction of this percentage of array resources dedicated to reconfiguration management. Our argument for this stems from an observation that virtual circuits are unlikely to scale in size with the technology. Rather, we would be more inclined to support more of them simultaneously, keeping essentially the same reconfiguration managment task and scaling its implementation slightly to account for the extra managment load. Rather than investing 100k gates of a million gate architecture, we could quite conceivably exploit 60k of those gates for virtual circuits and leave 40k for an expanded virtual circuit manager.

Gathering these requirements into an overall assessment of the performance to be met by a system supporting virtual circuits, we make the following assertions. To support configuration and interaction with virtual circuits, the supporting system should drive the configuration and circuit interaction interfaces at their core speed. In the case of the XC6200, the most advanced partially reconfigurable device available when this thesis was written in 2000, we must support a 32bit configuration port with a 40ns access cycle[2] with a corresponding raw data rate of 25MB/s. We can also consider our earlier requirement constraint of 10% of array resources dedicated to reconfiguration managment. For the XC6264, the primary target platform for the implementation work described in Chapter 5, the resource investment we would be considering for reconfiguration managment will be in the order of 8000 gates.

[2]40ns is the shortest cycle time in the product literature that the XC6200 configuration interface can be accessed with.

4.2.2 Fundamentals: The Swappable Logic Unit

The Swappable Logic Unit(SLU) [14] is the VC analogue of a page or segment in a virtual memory system. At the conceptual level, an SLU is a logic circuit capable of performing a given logic function, transforming its inputs into a set of function outputs. It has two key attributes: a fixed geometry implementation; and input and output interfaces that are fixed in structure and relative positioning within the overall circuit design. Three general models for SLU input and output are suggested: wired signals on its perimeter, dedicated registers accessible through the host device's programming interface, and active interface accesses driven by the SLU itself.

The practical management of an SLU has been described within the context of a benevolent VC operating system [12] and requires that some constraints are placed on the array resources that can be directly accessed by an SLU. Two general examples of such constraints would be limiting the influence of the SLU configurations to within its geometric bounding box, and preventing it from directly interacting with IOBs. The notion of privileged, system SLUs was introduced to accommodate instances where a VC application may have valid reasons for accessing "protected" resources. However, we should also note that the notion of privileged and protected resources is entirely conceptual, and enforced by the programmed VC operating system. The device architectures available in 2000 do not have explicit hardware support for protecting resources allocated to one SLU from another SLU.

4.2.3 The Sea of Accelerators Model

Figure 4.1(i) shows a visualisation of the sea of accelerators VC model. In this model, SLUs are entirely independent units of computation and have no direct interaction with each other. All communication with the SLUs is done by state accesses through the device's programming interface to registers placed at the

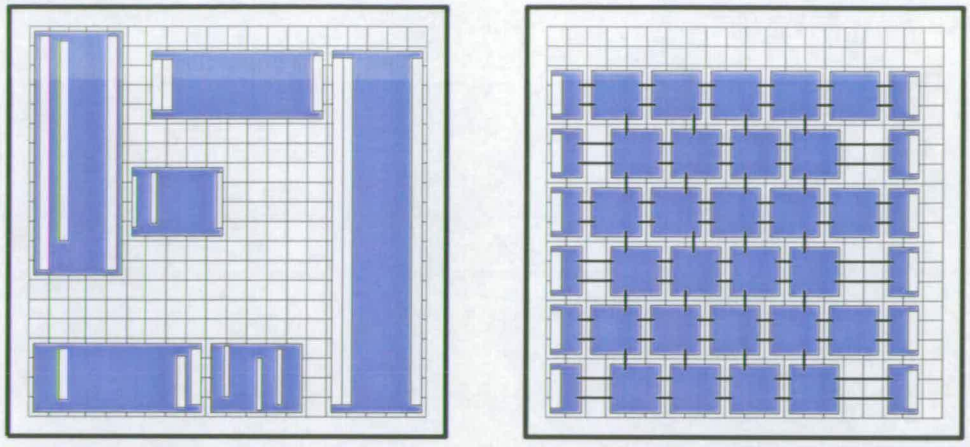

Figure 4.1: The two primary models of virtual circuitry: (i) The Sea of Accelerators and (ii) The Parallel Harness

SLU's inputs and outputs. In this model, the VC operating system has two main responsibilities: swapping SLUs on and off of the programmable logic; and facilitating access to the SLU input and outputs from the higher level VC application.

4.2.4 The Parallel Harness Model

Figure 4.1(ii) shows a parallel harness style VC model. Here, SLUs are essentially cooperating parallel processing elements. Whilst SLUs in the sea of accelerators never directly communicate, parallel harness SLUs are explicitly interconnected in a wired routing harness that is instantiated by the higher level operating system[3]. The use of explicit, wired routing implies that there is a regular structure to the overall parallel harness. FPGA routing resources tend to favour datapath circuitry which has regular interconnect wiring, making the mapping of irregular wiring structures generally more difficult.

At this point we should clarify the difference between the notions of a parallel harness SLU and a parallel harness circuit. In this thesis, we shall consider a

[3]In [16], the parallel harness interconnect strategy includes the deliberate abutment of SLU interfaces to facilitate interconnection. For the sake of this discussion, we will focus on the use of an explicit, routed harness since this provokes more challenges to the efficient implementation of the VC operating system.

parallel harness circuit to be the collective functionality created by the interconnection of many, typically homogeneous, parallel harness SLUs within the wired routing harness. That is, to the VC application, it is the compounded functionality of SLUs within parallel harness that constitutes the main unit of computation rather than computational features of the individual SLUs themselves. In Figure 4.1(ii), we can see that the parallel harness SLUs are bounded by a set of system SLUs which function as a wrapper to the whole parallel harness circuit. It is through this wrapper that the VC operating system would typically interact with the harness's constituent SLUs.

4.3 An Abstract VC Architecture

In an abstract sense, the virtual circuitry metaphor provides an intermediary between the program oriented world of the high level VC application and the circuit oriented world of dynamically programmable logic. Essentially, the VC system encapsulates the FPGA's fine-grained computational parallelism behind a runtime system interface used by the von-Neumann style, sequentially programmed high level VC applications. Implementing a mapping in an efficient manner is difficult. It is often the case that the implementation of the VC interface cannot preserve the potential performance gains that runtime reconfiguration makes available. We can present the three general causes of this performance drain:

- first, the narrow bandwidth peripheral bus interfaces in VC system architecture makes it difficult to interact with the programmable logic subsystem; and
- second, the standard programmable logic device interfaces make it difficult to interact with the programmable cell resources and the SLUs configured on them; and
- third, on a conceptual level the system must rapidly bridge between the

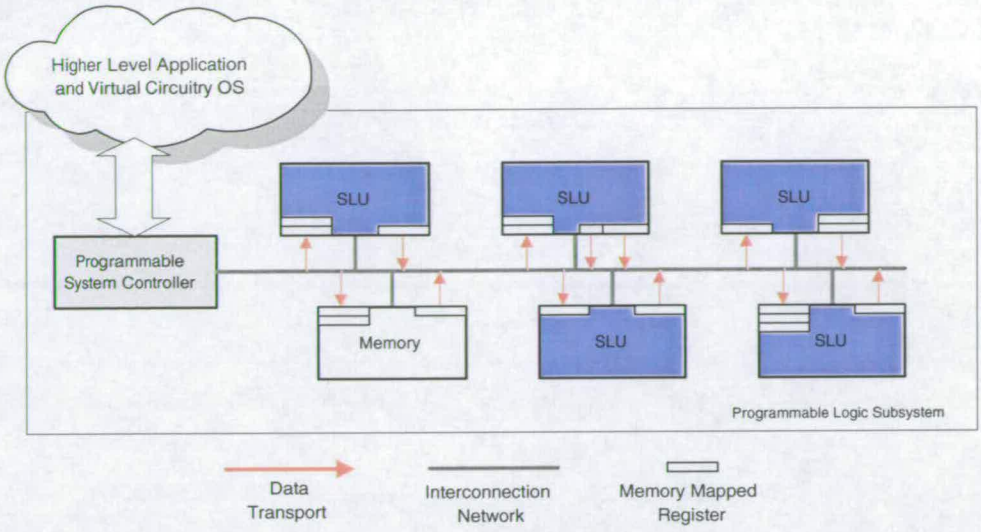

Figure 4.2: The Abstract Virtual Circuitry Architecture

domains of programs and circuitry because of the close couplings that VC applications try to establish. There is a significant degree of inertia that must be overcome each time the two domains interact.

In this section we consider the form and semantics of an abstract architecture that will address all three of these points. The architecture, shown in Figure 4.2, is implemented entirely within the programmable logic sub-system and supports the two virtual circuitry models that have been presented above. It has three constituent components: a set of SLUs with memory-mappable, register based interfaces; an underlying bus-style network that interconnects all the components in the abstract architecture; and, at its heart, a programmable system controller. It is the combination of a self contained implementation coupled with a particular style of programmable system controller that will let us overcome all three VC system performance issues.

We must clarify the concept of a SLU within the abstract architecture because the assertion that SLUs have register based interfaces appears, at first, to prevent the architecture from supporting parallel harness style VC. The abstract architecture SLUs are, at the conceptual level, somewhat abstracted from those

we described earlier. In particular, we do not rigorously apply the same explicit distinction between parallel harness and sea of accelerator SLU types. In the earlier discussion, the interface of a parallel harness SLU is formed by a collection of signal wires at the circuit periphery. However, what we considered as a parallel harness circuit, i.e. a collection of wired SLUs wrapped by register interface oriented system SLUs, constitutes a single "abstracted" SLU in the abstract architecture. Essentially, the abstract SLU definition captures the "task-level" computational completeness of the unit.

The programmable system controller has two main conceptual rôles: first, it supports the insertion and removal of SLUs to and from the current set of instantiated SLUs; and, second, it facilitates communication between the VC operating system and the instantiated SLUs. The underlying bus-style interconnect network is used to map the architecture components into the controller's memory map. The register oriented nature of each SLU interface is also significant as it reduces the complexity of mapping an SLU's IO ports into the system controller's memory map.

Although the controller supports two conceptual operations, the "memory-mapped everything" nature of the abstract architecture means that there is only one fundamental operation: the system controller implements data transports within its memory map. The programmable nature of the system controller is defined in terms of its ability to execute a sequence of data transports stored in a region of program memory. Indeed, Figure 4.2 contains an explicit memory interface SLU for this purpose[4]. We should also note that, in this basic implementation, the memory interface is single ported.

and function units. As a member of the TTA class, the higher level compiler for the abstract VC architecture would also have the opportunity to exploit such optimisations.

4.3.2 Alternative Architectures

This section addresses potential alternatives to the TTA-style abstract architecture we introduced above. There are two main alternatives: first, replacing the TTA style architecture with a mainstream instruction set architecture such as a microcontroller; and, second, state-machine oriented custom reconfiguration controllers. Section 4.3.3.3 discusses the advantages and disadvantages of state-machine oriented controllers and in this section we will focus on the use of alternative microarchitectures.

Our motivation for considering a full-blown embedded microcontroller to provide virtual circuit management is that it has the apparent advantage of having more computational power than our existing TTA approach. At first glance, the TTA world of the abstract VC architecture with its single move instruction appears computationally under-powered in comparison to instruction set microarchitectures: it effects only simple data transports within a memory map. In 2000, there are a large variety of microcontroller and embedded microprocessor architectures. It is difficult to give a succinct characterisation of their features: the diversity of the embedded control applications that such microarchitectures are used in means that it is common to see many variations and extensions on a basic instruction set family. The classic Intel 8051 embedded controller provides a good example: in 2000, there are easily tens of implementation variants of this basic controller.

In broad terms, a standard microcontroller will typically have intrinsic arithmetic support through an inbuilt ALU structure, conditional branching operations, and mechanisms for interacting with memories such as a variety of address-

ing modes and, quite possibly, DMA controllers. Interrupt handling is also an important feature of these architectures since, for example, their physical control applications must react to real-world stimuli.

For the specific application we are considering here – a microcontroller for virtual circuitry management – a pertinent question is: what does it cost to implement a microcontroller architecture within an FPGA architecture? Again in broad terms, the gate level cost of hosting a microcontroller is on the order of tens of thousands of gates. For example, the contemporary cost of a synthesised 8051 architecture is upwards of 20K gates, depending on the speed requirements of the final artifact. For this approximate budget we could conceivably implement a 10-20MHz microcontroller on what is considered a standard density, standard speed FPGA available in 2000. If we were to make an approximation of how this figure would rise in more aggressive and future technologies, microcontrollers on high-end future devices operating in the region of 100-150MHz are not unreasonable given the projected core circuit speeds within the prospective architectures.

Beyond the raw costs associated with the approach, it is also important to consider how a standard microcontroller interacts with the array it is reconfiguring and the SLUs it is interfacing with. A primary conceptual advantage of the abstract VC architecture is that its data transport oriented nature actually maps well to the fundamental tasks we wish to perform in a VC model. For the majority of our time we wish to manage the motion of data flowing between SLUs, the higher level VC operating system, and the configuration memory of the host FPGA. A data transport is fundamentally a very appropriate conceptualisation for this purpose. In the case of a microcontroller, the mapping between the SLUs resident on the reconfigurable logic and the configuration store of that same reconfigurable logic is not so clear. Microcontrollers, particularly in the RISC style, are predominantly register-operation oriented. To interact with SLUs residing on a FPGA like the XC6200, we could conceptually make them appear as registers

within the main architecture (the approach commonly used in research where reconfigurable functional units are added to a microprocessor). However, technically achieving this, even in the context of a relatively benevolent architecture like the XC6200 would be difficult given the basic interface abstractions presented by the two technologies.

In the context of contemporary devices, and in the context of our earlier discussion on the requirements to support virtual circuits, a microcontroller dedicated purely to virtual circuitry management is a substantial investment of array resources. By contrast, the implementation of the abstract VC architecture given in the next chapter requires a resource budget in the order of 4K gates. Based on this estimate, an abstract VC architecture implementation is more attractive in terms of raw resource utilisation. Furthermore, the relative complexities of the circuitry in the two approaches indicates that there is more scope to exploit raw performance increases in the device architecture and apply pipelining to the relatively simple circuitry of the abstract architecture.

The transport oriented style of the abstract VC architecture supports the conceptual mapping of SLUs into the sequential-style processing world of the microprocessor well. However, it is worth exploring whether aspects of the microcontroller architecture, such as branching, interrupts, or advanced memory interfacing and addressing could be usefully appended to the abstract VC architecture. For example, would it be useful to have an integrated adder circuit within the datapath of the abstract architectures circuitry and if so, how could it best be integrated?

There are two functionalities that appear to be of immediate interest in this case: first is the support of an incremental indexed addressing mode; and the second is the support of interrupt processing. Indexed addressing is of interest because it supports the style of sequential data accesses we expect to perform when transporting SLU configurations from memory to the configuration store

of the host device. Access to DMA controllers would be a natural technical evolution of this augmentation. Interrupt support is of interest for the purpose of implementing the active SLUs we considered within the SLU model earlier.

Beyond identifying these features as potentially useful augmentations to the pure architecture it is interesting to consider how we may actually perform the augmentation. One option is to erode the transport triggered nature of the abstract architecture and turn it into a very basic instruction processing architecture (this effectively means that we would add an instruction decoder to the abstract architecture. We would have integrated the additional instructions but since we access them explicitly by decoding an instruction operator, we could no longer consider the resulting architecture as purely transport triggered.). Alternatively, we could aim to preserve the TTA nature of the architecture and in the case of the indexed addressing extension, fold the additional features into the existing memory SLU that resides on the architecture's system bus. In comparing the two approaches we can see that there is an additional conceptual advantage to the TTA style abstract architecture: it inherently supports the addition of instruction logic by dynamically inserting SLUs on the system bus.

In summary, implementations of microcontroller style VC managers are technically feasible, but with current technologies and in the context of our earlier requirements to support virtual circuits, it is difficult to justify their resource utilisation. Furthermore, an integrated microcontroller is not simpler to interface with the host array architecture and the SLUs it contains. Despite this, there are architectural features of a microcontroller that, for performance reasons, may be complementary to the pure move-based abstract architecture. Furthermore, the nature of the TTA model supports the augmentation of these features without resorting to extending the core instruction set itself. This allows us to continue exploiting raw performance gains within the logic implementing the data transport and benefit from higher level compiler optimisations available to TTAs.

4.3.3 Self-modifying Circuitry

So far, we have just indicated that the immediate environment of the abstract architecture is a fairly generic programmable logic subsystem. We can now define the controller's environment more clearly and say that, in the scope of this thesis, the abstract VC architecture is contained entirely within a single host FPGA. In this context, the programmable system controller takes on the very interesting rôle of an *array resident* configuration agent and, at the abstract level, this involves mapping the host FPGA's configuration memory into its own memory space. In doing so, however, the system controller gains the interesting attribute of self-reference, and exploiting this self-referentiality to actively drive the host FPGA's programming interface gives the architecture the potential to support "self-modifying" circuitry.

4.3.3.1 Requirements for FPGA based Self-modifying Circuitry

Hosting the abstract architecture places three main requirements on the target FPGA architecture:

- first, the circuitry implementation of the system controller must be able to access the device interface logic of the host FPGA;
- second, since the circuitry effecting the configuration is resident on the same FPGA, the FPGA architecture must be partially reconfigurable; and
- third, the host FPGA must have an "open architecture".

The first two requirements are hard technical requirements that are essential to the system controller's implementation. The third, however, is more a conceptual requirement. Hosting a configuration agent within the target FPGA represents the closest, most intimate coupling to the device's programming interface that is possible. Since the programmable system controller interacts with the host FPGA at this level, it is important that, as designers of such a system,

we have a clear understanding of the host architecture's operation and nuances. A closed architecture prompts the designer to be either fairly conservative or risk potentially physically damaging the host FPGA by effecting configurations that create electrical contention within the device.

One extension to the first requirement, although not a strict requirement in itself, is that the host FPGA's device interface presents the FPGAs configuration store as a memory style interface. If this is the case, it reduces the semantic gap that must be bridged by the abstract architecture to integrate the host FPGA's configuration store into its own memory map. Specifically, it reduces the amount of interfacing logic required to map the host FPGA into the system controller's memory map.

In total, there are three conceptually distinct regions in the programmed controller's memory map. Earlier we discussed a region of generic memory for holding the programmed transport sequence that is executed by the controller. In this section we discussed the mapping of the host's configuration store into the memory map, enabling the architecture's data transports to implement circuit configuration and fulfil the first VC rôle. This leaves the fulfilment of the architecture's second conceptual rôle as a communication agent for instantiated SLUs. In terms of the abstract architecture, this rôle is facilitated by bus-style interconnect to the architecture's components. Specifically, integrating the host FPGA's state memory as the third region of the abstract architecture's memory map allows the controller's programmed transports to access the registers at the inputs and outputs of instantiated SLUs.

4.3.3.2 The Self Modification Taboo

Traditional software which has access to its program text and data segments has the potential for self-reference, and hence, self-modification. In modern software engineering practices, however, the exploitation of such properties is rare and taboo. For large software systems, this is a justified notion as the unruly appli-

cation of self-modification makes self-modifying software particularly difficult to debug. Indeed, it is common for processor architectures to reinforce this taboo through read only instruction caches. Significant cache penalties await programs which override the memory protection facilities of an operating system since modified sections of the text segment in the instruction cache must be flushed.

Striving for efficiency in a resource constrained environment is, traditionally, the main reason for exploiting self-modification. For example, the limited memory, storage, and processing time available in early computer systems justified the use of self-modification to gain increased code flexibility whilst limiting resource utilisation. VC systems in 2000 find themselves in an analogous situation to early software systems: FPGA device densities are still limited and the performance penalties associated with SLU instantiation and interaction are high. Therefore, the raw performance advantages to be gained from adopting self-modification as the technique for altering the configuration of a resident circuit, are particularly alluring.

4.3.3.3 Self-modification in related research

The concept, potential, and mechanisms of self-modifying circuitry using programmable logic was suggested in the literature [95, 108] we discussed in Chapter 2. The introduction of dynamic and partially reconfigurable FPGA architectures rekindled this interest. In particular, there are two notable conceptual systems: the self-reconfiguring processor [41] and the self-reconfiguring computer system [88]. We have also presented concepts, mechanisms, and novel properties of the self-modifying abstract architecture discussed in this section in the literature [32, 16]. Other work has advocated use of self-configuration within a framework for managing runtime reconfigurable designs [94] and, in subsequent literature, both the modeling and synthesis of controllers for self-configuration [87] was explored.

In 1999, a small number of applications of circuit self-modification have also

been presented in the literature, ranging from applications of genetic programming [96] to string-matching [97, 76]. Although we characterise the abstract architecture as supporting self-modifying circuitry, the intention of this thesis is not to pursue application case studies of self-modification in logic circuitry. It is nonetheless interesting to explore how McGregor and Lysaght [76], as an application of self modification, relates to our abstract architecture since they both apply the same technique on the same FPGA platform.

Specifically, this application attempts to take control of the XC6200's configuration port using the mechanism described in Chapter 5. When applied successfully, this places the XC6200 into a self modifying state. Two circuits are transferred onto the array by this stage, a circuit dedicated to controlling the reconfiguration process and the application circuitry (this is the circuit targeted by the reconfiguration process). The application circuitry implements bit level pattern matching on a serial datastream: it observes a bit serial stream of data arriving on a device pin and attempts to match sequences within that datastream against a specific match sequence that is specified though a second, separate bit serial stream. As the matching pattern changes, the application logic triggers the reconfiguration logic to reconfigure a constrained region within the application circuitry dynamically, adapting parts of the match circuitry to implement the matching sequence.

The application is notable for two reasons: first, it exploits a custom configuration controller circuit designed specifically to reconfigure the specific application circuit at hand; and, second, as a consequence of its customised nature, the reconfiguration circuitry itself generates the dataword that is used to alter the application circuitry. However, one significant limitation that the approach has is that it is designed specifically to effect only reconfigurations. The abstract VC architecture described above is designed to inherently support the reconfiguration of the host device and facilitate interactions with the SLUs that it has reconfig-

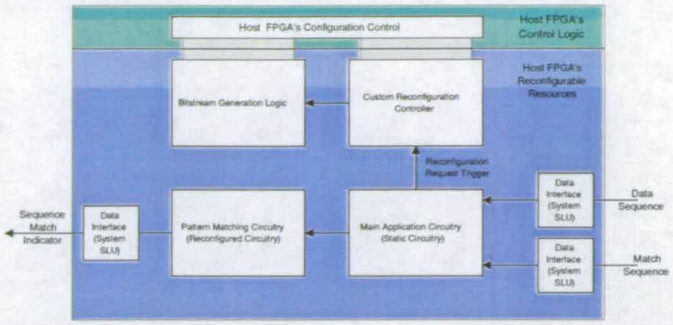

Figure 4.4: Organisation of the self-configuring pattern matcher[76]

ured. By contrast, the custom reconfiguration circuitry approach, as described, used explicitly routed signals to support communication between the application logic and the reconfiguration controller. Furthermore, this communication was solely for the purposes of expressing reconfiguration requests and not to interact with any registers within the SLU. Data communication within the pattern matching example essentially resembles a parallel harness circuit with data arriving through system SLUs that interface with application data ports physically wired to specific device IOBs. Figure 4.4 shows the general architecture of the application and its relation to the host FPGA's control logic.

There are some immediate comparisons that we can make between the approach in this application and the approach we will pursue through the abstract VC architecture described above. The first of these relates to the application specific nature of the reconfiguration controller. In [76], the authors describe the design methodology, modelling and synthesis approach used to generate the circuitry of the custom reconfiguration logic. For each application and reconfiguration schedule within that application, a new reconfiguration controller circuit is created to embody the reconfiguration task. By contrast, the abstract VC architecture's circuitry maintains a static size with respect to the reconfiguration schedule. Rather than generating bitstreams to embody alternative reconfiguration schedules, we encode them as sequences of executable software. However, one particular advantage that the custom reconfiguration controller approach has

is that, for a suitably constrained set of reconfigurations, it can generate the reconfiguration data internally. This avoids a potentially costly series of memory transactions, especially if the memory blocks are external to the reconfigurable resource.

An interesting question to raise at this point is which of the two approaches is most effective for supporting virtual circuitry. We can answer this relative to the context of the reconfigurable application we wish to support. The string matching application we have discussed above is an example of a small, well constrained reconfigurable application. The constrained nature of the application and, in particular, reconfiguration schedules are important as the size of the reconfiguration logic will grow with complexity of the reconfiguration schedule and the overall application. The custom reconfiguration controller approach may have performance benefits to gain in certain system architectures (particularly where there is a high cost to memory transactions), but the approach is more difficult to scale as applications grow in size.

Conversely, the abstract VC architecture has a constant size and performance[5] but the circuitry itself is not constrained to supporting any one application or reconfiguration schedule. We can argue that there is a cutoff point at which the static size of the abstract VC architecture's circuitry will be more attractive than the custom reconfiguration controller circuitry. An additional point to consider is that the abstract VC architecture, though its intrinsic support of SLU interaction, can be used for tasks other than supporting reconfiguration. The custom reconfiguration controller, on the other hand, remains a static investment of array resources that can only be offset if we have an application that demands a saturation of the reconfiguration port.

In terms of overall performance, the custom reconfiguration controller is attractive because we can dedicate as much circuitry resource to it as needed to

[5]in effect, size and performance are relatively constant since the architecture supports the dynamic use of other SLUs to accelerate the reconfiguration process.

ensure that it meets the timing requirements of the application. In performance terms, the generality of the abstract VC architecture is of concern since, as a rule, generality tends to dampen performance. However, implementations of the abstract architecture that are capable of saturating the reconfiguration interface of the host FPGA are conceivable (we will consider the performance potential of the abstract VC architecture in the latter sections of Chapter 7). Reconfiguration bandwidths available in the FPGA architectures in 2000 are unlikely to outstrip the performance of the abstract VC architecture. However, if that situation were to arise in future device architectures, then we would have the option of adopting custom reconfiguration controllers or timesharing the raw configuration interface between multiple instances of the abstract architecture.

Chapter 5 contains a discussion of the implementation of this chapter's abstract architecture. In it, we address the challenges of supporting self-modifying circuitry and may allude to uses of self-modification within the abstract architecture throughout later chapters.

4.4 Performance Enhancing Techniques for VC

Earlier in the chapter, we approached the subject of performance penalties associated with SLU swapping. Even with partially reconfigurable architectures, where we only need to configure the array resources required by the incoming SLU, the configuration cost is prohibitive for rapid, runtime reconfiguration. For early VC systems and applications, a significant part of this cost is associated with the limited bandwidth available in the loosely coupled co-processor system context. The abstract architecture discussed above, because of its intimate placement and relationship with its host FPGA, avoids this primary source of bandwidth starvation by tightly coupling the programmable system controller with the SLUs on the system bus. However, as we also noted in Chapter 2, bypassing the limited bandwidth peripheral bus of first generation co-processor architectures does not

completely remove the cost associated with SLU swapping since there are inherent limitations associated with particular styles of system interconnection architecture. We could argue that the bus-oriented nature of the abstract VC architecture is an example of such an inherent limitation. However, the abstract VC architecture presented is a fairly basic, conservative incarnation of a TTA. Comparing the abstract architecture with the generalised TTA architecture shown in Figure 4.3, we can see that there is scope for increasing the flexibility of the system bus into a system interconnect network.

In the sections below, we give an overview and discussion of different techniques that attempt to reduce some of the fundamental costs associated with SLU swapping. The majority of these techniques were introduced to combat the bandwidth limitations of first generation VC co-processors. However, they are equally applicable in the context of the abstract VC architecture. Indeed, some of the techniques are particularly effective for increasing the parallelism available within the bus-oriented abstract architecture.

4.4.1 Partial Reconfiguration

An implicit assumption in the discussion so far is that partial reconfiguration is the mechanism that replaces configuration granularity at the device level with configuration granularity at the SLU or, conceptually, task level. However, the techniques in the next two sections exploit partial reconfiguration to reduce the amount of SLU configuration data that must be applied between successive SLU configurations.

4.4.1.1 Incremental Differences

Rather than applying the complete SLU bitstreams, we can exploit commonalities between the array features that two different SLUs use and only apply the incremental difference between the two. At best, the incremental difference will be much smaller than the second SLU bitstream and, at worst, we will have to

resort to applying the entire SLU configuration[6]. The application of incremental differences was considered as part of the framework in the original VC model descriptions [12, 13] and a toolset for calculating incremental differences exists [68]. The Virtual Hardware Handler [104] also calculates the inverse incremental differences, allowing the system to move backwards and forwards between SLUs in the configuration schedule.

4.4.1.2 Runtime Reconfigurable Routing

In most cases, partial reconfiguration of SLU circuitry is focused on changing the configured functionality of the array cells used by the SLU and routing is considered a second-class object. The design of partially reconfigurable circuitry typically holds the wired routing of the circuit as static and focuses on making constrained changes to the array cells. Applications of partial reconfiguration that specifically target the wired routing of an SLU are rare.

Runtime reconfigurable routing [16] for parallel harness style VC can be advocated as a means of increasing the flexibility of the wired routing harness synthesised by the VC operating system. The earlier discussions imply that the parallel routing harness is large, and statically defined for the duration of the parallel harness circuit's instantiation. In [16], the concept of a reconfigurable switching fabric is presented. The fabric is essentially a complete parallel routing harness with the exception that a set of configurable switching points are defined. Changing the configuration at the switching points allows the interconnection topology of the harness to be rapidly altered. The form and distribution of the switching points must be carefully chosen to avoid making the partial reconfiguration of the switching fabric too costly. An application case study of a configurable crossbar switch [34] demonstrates an effective implementation of a 32×32 crossbar.

[6] Applying this technique at runtime assumes that it takes zero time to discover the commonalities between SLUs and generate the difference bitstream. If there are no common features, we also assume that it takes zero time to determine this. This is impractical in reality so incremental differences are calculated 'offline', using a pre-determined configuration schedule.

The cost of the flexibility in the configurable switching fabric is the possibility that it will serialise the SLU communication that can occur within the harness. That is, SLUs have a higher chance of being isolated from the harness when the fabric is configured in a particular way. However, this may actually be beneficial and, in some situations, actually extend the applicability of virtual circuitry. For example, consider two SLUs that require access to the same, contested resource on the cell array. The arbitration that is required can be achieved automatically by swapping the SLUs on and off of the host cell array – the act of swapping itself serialises access to the contested resource. However, we cannot rely on standard SLU swapping to implement arbitration if the SLUs require a tightly interleaved access schedule: the cost of the frequent and rapid swapping of SLUs would soon overwhelm the system[7].

Runtime reconfigurable routing can be used to alter the configuration of a switching fabric that connects either SLU to the contested resource. We should note that we are assuming the spatial costs of having both SLUs simultaneously present on *some* part of the array can be justified. The form of the switching fabric supports the desired serialising effect and we apply partial reconfigurations to it to determine which of the two SLUs has access to the contested resource. The configuration cost of changing the fabric's switchpoint configuration is much less than that of swapping an entire SLU. The rapid reconfiguration of the fabric is, therefore, less likely to overwhelm the system in the same way that full SLU swapping would.

4.4.2 Partial Evaluation and Constant Propagation

Partial evaluation [99] and data folding [40] are synonymous references to design-time optimising techniques based on constant propagation. They produce specialised instances of SLU circuitry based on the constant propagation of a semi-

[7] We assume that a single context device architecture is being used here. A similar serialising effect without the configuration performance penalty could be achieved by placing SLUs on independent contexts of a multicontext FPGA and rapidly swapping between them.

static input. The main benefit of partial evaluation is that a specialised SLU will generally operate faster than the non-specialised version. However, partial evaluation is also interesting in the context of this discussion because specialised SLUs are typically also smaller. We should clarify our notion of smaller in this instance since the geometric area of a partially evaluated circuit depends on the style of partial evaluation that is used. For example, the partial evaluation engine described in [77, 78] produces specialised SLUs that take up the same geometric area, whilst the approach outlined in [19, 52] may also reduce the geometric area of the specialised SLU. In both cases, there is a reduction in the size of the SLU's configuration bitstream since the specialised circuit requires less of the array resources than the generalised version.

4.4.3 Configuration Compression

The amount of raw data that must be transferred to an FPGA in order to instantiate a particular SLU can be reduced by various data compression techniques. Specifically, these techniques use data compression algorithms that exploit any regularity within an SLU's raw bitstream data to reduce its overall size. The performance of the VC system using compressed bitstreams may then increase because less data needs to be transferred to the target FPGA over the slow co-processor interface. However, this assumes that the costs associated with decompressing the bitstream are sufficiently low. Decompression is typically done by decompression circuitry that has either been especially configured onto the array for that purpose, or actually forms part of the underlying device architecture.

A compelling example of configuration compression using the wildcarding facilities of the XC6200's FastMap interface is given in [49, 65]. Wildcard based compression tries to identify writes of the same, or suitably similar, data words to multiple distinct addresses. Rather than perform, say, four individual writes, the sequence is re-encoded as a single wildcarded write that would simultaneously

transfer the data word to the appropriate configuration memory locations. Since the wildcarding hardware is actually part of the underlying device architecture, decompression is essentially free.

The wildcarding approach is device architecture specific, but other work has considered the use of general data compression algorithms from the software domain [50]. For example, standard Huffman or Liv-Zempel compression of the bitstream may be used to reduce its raw size. On the target FPGA, however, we must instantiate a corresponding Huffman decompression engine. The outputs of the decompressor would then be fed to the host FPGA's programming interface. To increase the decompression performance, it is even conceivable that the Huffman decompression circuit could itself be runtime reconfigurable [17]. It seems clear that as the compression and decompression schemes become more complex, the investment in managing the compression also increases. This must therefore be balanced to prevent the potential performance gains from applying compressed configurations being lost in the complexity of the decompression scheme.

4.4.4 Configuration Prefetching

In the discussion so far, we have presented a variety of techniques that try to reduce the amount of configuration that must be done to instantiate an SLU. Configuration prefetching [48] takes an entirely different approach and attempts, instead, to hide the reconfiguration penalty behind an ongoing "useful" computation. In the context of this chapter, the VC application code issues a non-blocking prefetch request for an SLU to the VC operating system before it plans to interact with it. The SLU would be pre-fetched by the operating system whilst the application continues processing in parallel. The main challenge for configuration prefetching is to determine the best point in the application code to issue the prefetch request or for the operating system to guess the best point.

4.4.5 Configuration Interleaving

Configuration interleaving is related to configuration prefetching, although it was introduced independently [32] as an abstract architecture specific technique. Configuration interleaving takes advantage of the transport programmed, memory mapped nature of the abstract VC architecture to mix transports from different architecture rôles at the granularity of a single transport. For example, one possible sequence of transports executed by the programmable system controller could begin with a single transport configuring part of SLU a, immediately followed by a transport configuring SLU b, which is, itself, followed by a transport for interfacing with a previously configured SLU.

Configuration interleaving is important in the context of the VC abstract architecture because the architecture typically has only a single access port to the host FPGA's programming interface. This prevents the parallel configuration and SLU interaction that is implied for configuration prefetch. However, through the fine granularity data transports used in interleaved configuration, we can implement an approximation of the parallel configuration used in configuration prefetching. Indeed, a real-time embedded-system application could exploit knowledge of hard deadlines and real-time priority scheduling techniques to allow gradual variations in the proportion of abstract architecture data transports that are applied to different application tasks. For example, more transports can be dedicated to configuring a particular SLU as its configuration deadline approaches.

4.4.6 Analysis

From the discussion above we can see that there is potential to apply a number of techniques to offset the performance penalty associated with SLU swapping. In this section we approach the techniques from a critical standpoint with the aim of prioritising them in the order that they are most likely to be effective for

general VC applications. Below, we consider each of the techniques in increasing order of desirability.

Not all of the techniques described earlier can be applied in the same application without one adversely influencing another. For example, partial evaluation and incremental differences are not complementary techniques. The application of incremental differences relies on similarities being present within the circuitry but the partial evaluation, as described in [99], actually produces successive configurations of a circuit that are structurally different. Also, although the general concept of partial evaluation is attractive, it requires a non-trivial processing effort and time, relative to the timescales within which we will reconfigure circuits, to generate the optimised, partially evaluated circuits. Using the technique we can clearly generate smaller circuits with higher processing capability, but in the situation where circuits must be specialised rapidly according to a particular changing parameter, the investment required to actually apply the technique would overwhelm the reconfiguration schedule.

Reconfigurable routing, like partial evaluation, is a potentially powerful technique but it has a major conceptual limitation: routing is considered a second class object by circuit design methodologies and their corresponding tools. Nevertheless, the fundamental technical features exist within reconfigurable devices such as the Xilinx XC6200 to implement reconfigurable routing by, for example, rapidly altering the configurations of switching points in a wired switching fabric. However, without a clear way to describe routing as a first-class object within the design framework for reconfigurable systems, the large amount of implementation effort required to deploy the technique will offset any benefits available to general VC applications.

Prefetching and incremental differences are, potentially, complementary techniques which are both based on having advance knowledge of the configuration schedule. Of the two, we can argue that incremental differences are much more

relevant to contemporary reconfigurable systems. Prefetching hides the latency associated with loading a particular SLU behind other system processing and relies on statistical profiling of the application to determine as close to optimally a point at which loading should begin. However, the arguments for investing such offline efforts to hide loading latency are waning as physical device densities in FPGAs increase. The latency associated with transferring the SLU bitstream into the FPGA's configuration memory is less and less relevant as increasing numbers of SLUs can be made resident on high density architectures. Whilst earlier architectures had difficulty holding an application's working set of SLUs, new architectures allow the full set to be accommodated within the array without repeatedly prefetching. Furthermore, as the bandwidth to the configuration interface increases, the cost of applying an incremental difference to a circuit will become less significant. In the instances where we still cannot accommodate the entire working set, rapidly modifying resident circuits with incremental configurations over a high bandwidth configuration interface involves less effort, and is simply more generic over multiple applications than profiling and prefetching.

From the reasoning and discussion above, and to return to our aim of prioritising the techniques according to their effectiveness, we can now advocate that partial evaluation is the least effective technique for VC applications. This is followed by runtime reconfigurable routing and then prefetching. The final technique we considered, and the one we consider most effective, is the application of incremental differences.

4.5 Sequential Algorithmic VC

We discussed two VC models earlier in this chapter. This section introduces a third, new model of virtual circuitry produced as part of the original research of this thesis. The fundamental feature that differentiates this third, new model and the two other VC models is the restriction they place on the way that VC SLUs

may interact. In the sea of accelerators, there is absolutely no interaction between SLUs whilst, at the opposite end of the spectrum in the parallel harness, there is a high degree of very tightly-coupled interaction between SLUs. In the third model of virtual circuitry, the "Sequential Algorithmic" model, SLUs are neither tightly-coupled nor completely independent. Instead, we use the VC abstract architecture to facilitate a *flexible* harness of loosely-coupled, co-operating SLUs that are configured within the cell array. The programmable system controller has the ability to effect data transports in any region of its memory map. We discussed earlier how programmed transports can effect configuration, how programmed transports can effect communication, and equated these abilities to the abstract architecture's two fundamental rôles of configuration agent, and communication agent.

The sequential algorithmic model taxes the full range of the abstract architecture's facilities to cast it in the third rôle of *computation* agent. As a computation agent, the architecture is responsible for implementing the loosely-coupled interactions between the instantiated sequential algorithmic SLUs. In the operational terms of memory-mapped data transports, this rôle involves the *rapid* transport of data within the region of the abstract architecture's memory map that contains the host FPGA's device state. Each transport within this region has the effect of briefly interconnecting the output of one SLU to the input of another. The sequential algorithmic model's flexible harness is actually a programmed sequence of transports executed by the abstract architecture in its rôle as a computation agent. Essentially, while the parallel harness has explicit and fairly static wired routing between SLUs, the SLUs of the sequential algorithmic model are interconnected by dynamic software routing. Just as the higher level operating system is charged with supplying the wired routing of the parallel harness, it is also responsible for supplying the programmed transport sequence for implementing the flexible harness.

The system controller applies its programmed data transports in a sequential manner. The serialised nature of this execution means that interaction within the sequential algorithmic model's flexible harness is also serialised[8]. In practice, we can envision modifications to the abstract architecture that would make SLU interaction within the flexible harness increasingly parallelised. For example, a multi-ported interface to the host FPGA's programming interface would allow a parallelised system controller to effect simultaneous data transports within its memory map. The FastMap interface of the XC6200 has a number of technology specific features that we could exploit to parallelise interconnections in the flexible harness. Specifically, the XC6200's wildcard mechanism also affects writes to the FPGA's state memory and would facilitate multicast-style transports. The map register mechanism could also be used to implement 'bin-packed' data transports in which two data operands are transported within the one data word. The map registers would define the appropriate distribution of the separate data operands to the target SLU's register interface.

Explaining the "algorithmic" component of the sequential algorithmic model name requires us to look more closely at the programmed transport sequence that implements the flexible harness. The programmed sequence could be just that: an enumeration of elaborated data transports that interconnect a defined set of SLUs, in a defined order. An alternative interpretation of the flexible harness transports is that they constitute an *algorithm* that actually consumes the results from some of the operations triggered by its transports. These results influence the interconnection sequence according to the particular algorithm encoded within the flexible harness's transport sequence.

[8]It is this attribute of the abstract architecture that inspires the "sequential" component of the model name.

4.6 Summary

This chapter began with a comprehensive discussion of the two main VC models and an abstract architecture that would support them. From there, we explored some of the auxiliary techniques that are deployed to make SLU swapping more tractable and closed the main body of the chapter by discussing a third, sequential algorithmic VC model, motivated by the abstract architecture.

In Chapter 3 we noted that programmable logic device interfaces have evolved and are specific to a class of applications. In this chapter we have, essentially, presented three general variations from the class of virtual circuitry applications. The next three chapters will present an implementation of the abstract architecture introduced in this chapter. We then show that this implementation is also an instance of Chapter 3's flexible programmable logic interface.

Chapter 5

The Flexible Ultimate RISC

In this chapter we present the design and implementation of the Flexible Ultimate RISC, an instance of the abstract microarchitecture introduced in the previous chapter. We tackle this in two main sections:

- First, we introduce the original Ultimate RISC architecture and present detail of its design and operation. This includes a description of the challenges of implementing even the simple Ultimate RISC architecture on the target XC6200 FPGA architecture.
- Second, we present details of the Flexible Ultimate RISC which is an evolved version of the simple URISC that is capable of autonomous self-modification. This discussion focuses on the challenges of facilitating self-modifying circuitry on the target FPGA architecture.
- Third we present details of the design, development and general programming and runtime environment of the Flexible Ultimate RISC.

5.1 The Ultimate RISC(URISC)

The Ultimate RISC(URISC)[57] is a minimal processor architecture with only one instruction: *move memory to memory.* On each instruction cycle, a single word in memory is moved from one location to another. Computation is done by migrating devices onto the system bus, then mapping the input-output registers

Chapter 5

The Flexible Ultimate RISC

In this chapter we present the design and implementation of the Flexible Ultimate RISC, an instance of the abstract microarchitecture introduced in the previous chapter. We tackle this in two main sections:

- First, we introduce the original Ultimate RISC architecture and present detail of its design and operation. This includes a description of the challenges of implementing even the simple Ultimate RISC architecture on the target XC6200 FPGA architecture.
- Second, we present details of the Flexible Ultimate RISC which is an evolved version of the simple URISC that is capable of autonomous self-modification. This discussion focuses on the challenges of facilitating self-modifying circuitry on the target FPGA architecture.
- Third we present details of the design, development and general programming and runtime environment of the Flexible Ultimate RISC.

5.1 The Ultimate RISC(URISC)

The Ultimate RISC(URISC)[57] is a minimal processor architecture with only one instruction: *move memory to memory.* On each instruction cycle, a single word in memory is moved from one location to another. Computation is done by migrating devices onto the system bus, then mapping the input-output registers

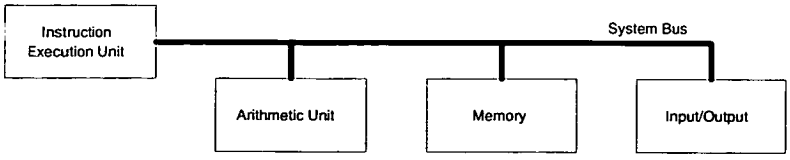

Figure 5.1: A minimal URISC Implementation

of those devices into the memory space of the URISC processor core. For example, the datapath of the URISC has no explicit arithmetic-logic unit(ALU); instead, a series ALU-like components reside on the system bus with their registers mapped into the memory space of the URISC core. Operands and results are then moved to and from the memory addresses which correspond to the registers of ALU components, as a means of performing arithmetic computations.

The URISC was first introduced as a novel example of the reduced instruction-set philosophy taken to an extreme and, given its simplicity, was not intended to be competitive with mainstream RISC and CISC processors. Despite this, it was noted in subsequent literature that, with some slight modifications to the basic architecture[43], it is possible to implement a more powerful URISC which fared better against other microarchitectures executing the same benchmark.

There are four main components in a minimal URISC implementation, shown in Figure 5.1: the heart of the URISC itself, the Instruction Execution Unit; a single-ported memory for holding program instructions and operands; a collection of ALU fragments to support computation and program control flow; and a general input-output interface to allow the processor to interface with peripheral devices. Faster implementations of the URISC are conceivable by increasing the number of memory ports and pipelining the IEU, but for the sake of this discussion, we will focus on a basic implementation.

5.1.1 The Instruction Execution Unit(IEU)

The IEU, shown in Figure 5.2, is responsible for implementing the move instruction and we can decompose it into two basic parts: a processor datapath and

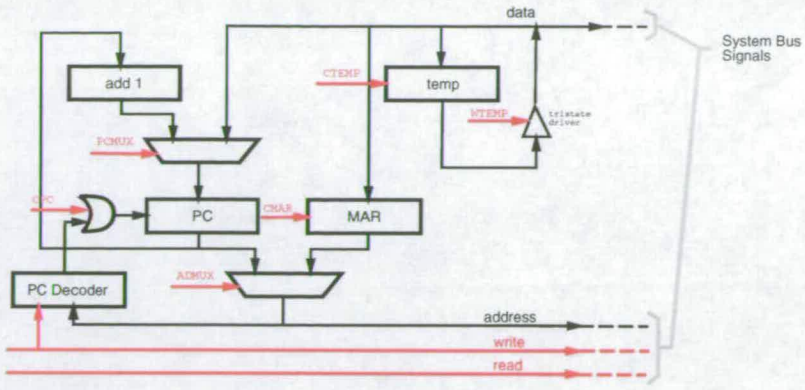

Figure 5.2: The Ultimate RISC Datapath

processor control logic. The IEU datapath is particularly lean. It consists of only three 32-bit registers, a 32-bit incrementor, a simple address decoder, and some basic 32-bit multiplexors. The URISC system bus contains a bidirectional data-bus and a uni-directional address bus which are both 32 bits wide. Some auxiliary control signals for read and write control of system bus elements also form part of the system bus.

From left to right, the three registers in the IEU operate as a program counter (PC), a memory address register (MAR), and a 'temporary' register. The program counter always contains the address of the next part[1] of a move instruction that will be fetched from memory. The MAR holds the source address, and then the destination address of the operand that will be transferred by the currently executing move instruction. The temporary register is used to hold the operand between the time that it is fetched from its source address to the point that it can be written to its destination address. A tristate buffer is present between the output of the temporary register and the data bus to facilitate bidirectional communication on the data bus.

The control program for the URISC is also quite simple. Since move is the only instruction no operand decoding is necessary. Within each move instruction cycle, there are four discernable micro-cycles. Table 5.1 gives a pseudocode

[1]We are assuming that a move instruction consists of two separate memory words.

Microcycle	Activities
1	$addr \leftarrow M[pc]; pc \leftarrow pc + 1$
2	$temp \leftarrow M[addr];$
3	$addr \leftarrow M[pc]; pc \leftarrow pc + 1$
4	$M[addr] \leftarrow temp;$

Table 5.1: Control Microprogram for basic URISC implementation

style specification of the activities undertaken at each instruction 'microcycle'. Each row in the table corresponds to a single microcycle. The first microcycle fetches the source address for the move from the address contained in the program counter which is then incremented. The second microcycle then fetches the datum contained in that source address and latches it into a temporary register. In the third microcycle, the destination address is fetched from the program counter address which is subsequently incremented. The final microcycle transfers the datum held in the temporary register to the destination address.

There is one notable limitation of the control program as it is presented: there is no way to actively change the contents of the program counter, so it is not possible to branch. This is solved by simply making the PC addressable and then moving a branching address into it. The address decoder present in the IEU datapath is used for this purpose. When the MAR outputs the PC address[2] onto the address bus, and an appropriate control signal is asserted by the control program, the PC can be written. Otherwise, the PC takes its next value from the output of the IEU incrementor which is, itself, in a latched feedback loop with the output of the PC.

In total, there are eight control signals used by the control logic to operate the IEU datapath. These signals are listed and their rôles explained in Table 5.2. The actual implementation of the control microprogram with respect to these control signals is shown in Figure 5.3 as a timing diagram. In the single-ported memory implementation of the URISC, 16 core clock cycles are required to implement

[2]by convention we use address zero for the program counter.

Signal	Rôle
PCMUX	Controls the multiplexor which decides whether the next value to be stored in the program counter comes from the output of the incrementor, or is the operand that was fetched from the source address.
WTEMP	Controls the tristate buffer allowing the output of the temporary register to be asserted on the bidirectional data bus. This signal is asserted in the final microcycle when we wish to write to the destination address.
CTEMP, CMAR, and CPC	All three of these signals are used to control when registers in the IEU datapath may latch in the data in the operand data that has been presented on the data bus.
ADMUX	The ADMUX signal is used to define which IEU register may drive the address bus. The PC is driven onto the bus in microcycles one and three, whilst the MAR drives the address bus in microcycles two and four.
READ and WRITE	These signals form part of the system bus that interconnects the IEU with the system bus elements. They indicate whether the current operation being performed over the system bus is a read or a write.
ADDR and DATA	These signals are also part of the system bus and communicate the address and data word being transported between system bus elements.

Table 5.2: Control signals used in the control path of the original URISC

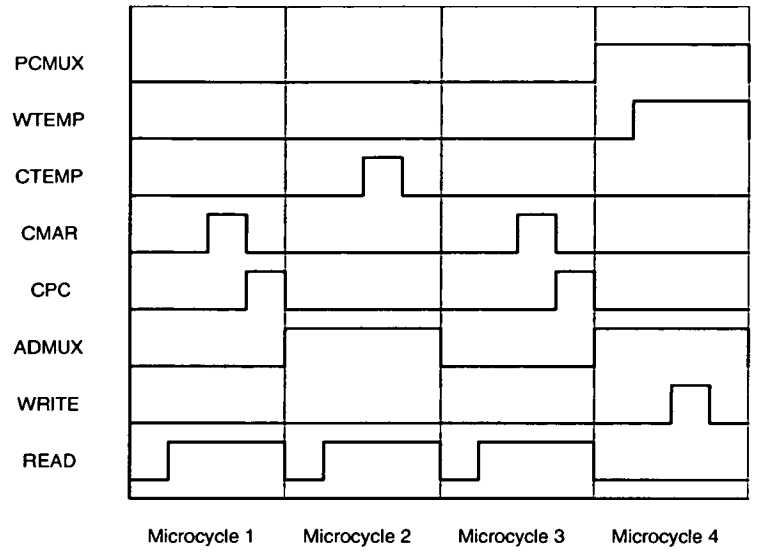

Figure 5.3: Control Waveform for the basic URISC

a single move operation, 4 cycles for each instruction microcycle. One notable point in the timing diagram is the assertion of PCMUX for the entire fourth microcycle. The signal is held high to facilitate writing to the PC if the address currently driven onto the address bus matches the PC address. The PC decoder present in the IEU datapath uses all system bus address signals but only asserts its output when the WRITE signal is also asserted.

5.1.2 URISC Programming

Generally speaking, a URISC program is a sequence of move instructions. Technically, each move performs the same fundamental operation, but logically some moves effect computation to further the current calculation directly. Other moves effect computation to manage the program control flow.

5.1.2.1 Branches

Implementing branches in a URISC program can sometimes be complex. The simplest form of control flow branch in a URISC program is an unconditional jump and is simple to implement. Since the jump address is static, we just move the jump address into the memory mapped program counter.

In a conditional branch, however, the final jump address depends on the result of some conditional test. Generally, processors have dedicated instructions which perform the conditional test and have the side-effect of altering the program counter based on the result of that test. In the URISC there are no such instructions. Instead, we use a memory mapped condition code register in an ALU fragment to bias the destination address of an unconditional jump. This allows the destination of the jump to be offset by the equivalent integer value of the boolean result contained in the condition code register. In this manner, it is possible to skip at least one move instruction following the unbiased jump address. If the instruction either of the possible jump addresses was itself an unconditional jump then effectively, based on the result of the conditional test, we would either take the unconditional jump or skip it entirely.

To make this approach work, we adopt a convention where the boolean truth value for a conditional test is an integer multiple of the size of a single move instruction, measured in words, and the boolean false value is integer zero. This is important as it allows us to guarantee that the biased jump address is still aligned with the start of a move instruction. For the URISC implementations described in the later sections of this chapter, each move instruction will comprise two 32-bit addresses and, therefore, occupies two consecutive words in memory. The integer equivalent to the boolean truth value in this situation, therefore, would be the integer value two[3].

5.1.2.2 Addressing modes

One primary characteristic of a RISC architecture is its lack of complex addressing modes. All addressing in the basic URISC implementation is absolute. Although we do not have inherent access to immediate or indirect addressing modes, we may again adopt programming conventions to emulate these two useful modes.

[3] In some circumstances this is not the case: Section 6.3.2.1 contains examples of truth values that must be larger than two.

Immediate addresses, for example, can easily be converted into absolute addressing of operand data placed in a predefined location. The assembler, described later in this chapter, automatically converts immediate addresses in the source code to absolute addresses of literal values pre-placed in a data table.

Indirect addressing is slightly more difficult to emulate and requires that we use self-modifying program code. Pointer dereferencing is a main motivating example of indirect addressing. The arguments against the use of self-modifying code do not particularly apply here since we are not using self-modification to radically change the program structure or behavior. Instead, the technique is limited to use in a narrow and well defined problem instance.

We can emulate indirect addressing for both the source and destination addresses of a move instruction. For example, we use two absolute move instructions to implement a single, indirected read. The destination address in the first move instruction is actually the address of the following move's source address. When we execute the first move, it overwrites the source address of the following move instruction with the address of the actual data we wish to read. Essentially, we are dynamically synthesising a customised absolute-addressed move instruction, one instruction cycle before it is required. If we want to perform an indirectly addressed write, then we alter the destination address of the first move so that it overwrites the destination address of the second move. If we wish to indirectly address both source and destination, then we need three consecutive move instructions. The first two modify the source and destination addresses of the third which then performs the desired transport.

5.1.3 Challenges of a XC6200 URISC Implementation

In this section we describe an original, technical implementation of the URISC IEU on a XC6200 FPGA undertaken as part of the research programme for this thesis. The components of the URISC architecture that were implemented at this

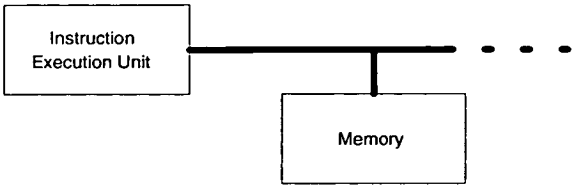

Figure 5.4: Basic architecture of the initial URISC implementation on the XC6200

stage in the research programme are shown in Figure 5.4 and there are two main problems that we are going to consider: a means of avoiding or limiting the use of tristate logic in the URISC core; and, second, bootstrapping the control logic of the IEU so that the processor is immediately capable of processing at the end of the configuration process.

5.1.3.1 Lack of Tristate Signalling

There are very few tristate resources available to logic on the XC6200 but, in the URISC IEU datapath shown earlier, a single tristate data bus is used to connect the URISC with each of its system bus components. The general routing resources of the XC6200, as we discussed in Chapter 2, are based on multiplexors and directed routes. The only tristate resources that are available to XC6200 circuits exist within the device's input-output pads. Physically, it is possible to connect a single bidirectional signal to a device pad and, when that signal is routed through the associated IOB, it is separated into independent read and write nets. The device pad contains a single tristate driver whose control signal is available to user circuitry through the pad's IOB. This allows the write signal to be driven onto the physical device pin and, hence, interact with a bidirectional physical bus.

For the URISC implementations we describe in this chapter, the intention is not to explicitly wire each SLU to the FPGA implementation of the URISC core. Instead, as we alluded in Chapter 4, the XC6200's FastMap interface is used to allow us to both configure and interact with the SLUs implemented on the logic

array. Essentially, we use the FastMap interface as a dedicated, memory-style interface to the system bus computational elements. There is no physical tristate data bus that the URISC must explicitly manage[4]: the FastMap itself implements the SLU section of the system bus.

We cannot avoid a direct, physical, tristate interface between the URISC core and the memory subsystem since memory is provided by physically interfacing with external memory components. The data bus of the memory interface is inherently bidirectional. Our challenge is therefore to use the tristate buffers in the device pad and the facilities of the IOB to manage this by providing separate read and write ports into the memory system. The URISC 'WRITE' control signal triggers the tristate driver within the device pad whenever a write is made by the URISC to the device memory. An updated IEU datapath in Figure 5.5 shows two temporary data registers which interface with the separated read and write interfaces of the memory system. An updated control timing diagram is also given as Figure 5.6.

We use two temporary registers since all data arriving from external memory must pass through the temporary incoming register. In the original system, the temporary register only ever held the operand being moved; the inputs of each register in the core had an individual connection to the raw data bus. In the adapted design, we hide the raw data bus behind the temporary incoming register and, hence, avoid connecting the inputs of core registers indiscriminately to IOBs. This is useful since we can use it to abstract the details of more than one memory interface away behind a single temporary incoming register.

However, the temporary register is now successively overwritten in the first three microcycles. We must preserve the data to be written during microcycle four by transferring it to a secondary 'outgoing' temporary register at the end of microcycle two. If this was not done, the transport datum would be overwritten

[4]although it is acknowledged that, internally, the FastMap data bus is also implemented as a collection of tristate data lines.

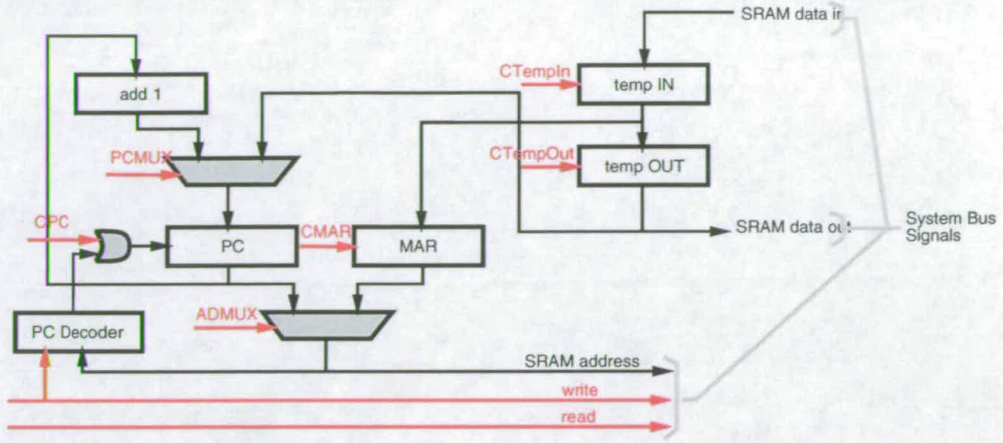

Figure 5.5: The URISC XC62000 Datapath

by the destination address arriving in microcycle three. A side effect of this arrangement, now noticeable in the control timing diagram, is that the CMAR control signal is activated one clock period later. This is because it takes one clock cycle to latch the value on the SRAM data bus into the temporary incoming register. Only after that cycle has passed can the incoming data be moved to the appropriate destination register.

5.1.3.2 IEU Control Implementation

One of the primary requirements of our URISC implementation is that it is designed to be autonomous. Whilst we may ultimately communicate with the external environment, we must not rely on the services of an external host to initialise or activate the URISC. This places an important requirement on the bootstrapping of the URISC itself. That is, from the point that we complete configuration of the XC6200 with the URISC bitstream, the core must become active immediately and begin processing its first instructions from memory. This should be done without the direct influence of an external source like a system's host processor. In short, the URISC implementation must be self-initialising and self-activating.

Each of the IEU control signals for the URISC is implemented as a 16-bit circular shift register. Each bit in the shift register maps to one state of that

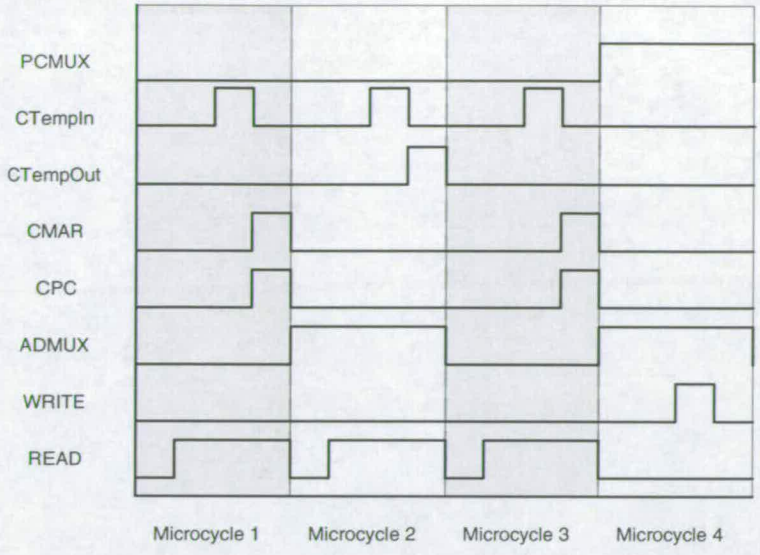

Figure 5.6: The URISC XC62000 Control Timing Diagram

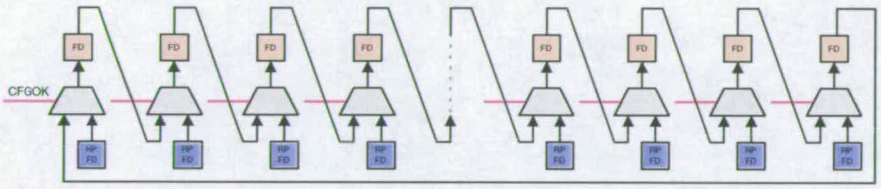

Figure 5.7: Self-initialising and self-activating control logic shift register

signal at a given instruction clock cycle and we tap the register at an appropriate point so that the register output at that point matches the control waveform. This repeats *ad infinitum*. The main challenge that we must address, however, is how to inject the waveform values into the register before the main circuitry becomes active and in such a way that the outputs of each shift register are aligned and synchronised to the control waveform.

Figure 5.7 shows a schematic representation of an URISC control shift register. In this figure `FD` components are d-type flip-flops and `RPFD` components are *register protected* d-type flip-flops. Both types of register are synchronised to the same global clock source. When a cell in the XC6200 architecture is placed in protected mode, its register is isolated from the remaining cell logic and is only sensitive to an underlying control input from the FastMap interface. Therefore, a `RPFD`

can only be modified by the XC6200 FastMap interface. It is possible to embed FastMap writes inside a configuration bitstream such that a RPFD can be initialised during the configuration process. This requires, however, that the clock source for the RPFD being initialised is active during configuration.

Each control shift-register, therefore, is shadowed by a RPFD register which is initialised during configuration to contain the appropriate control waveform values. The outputs of each RPFD are connected to the inputs of a corresponding shift register FD. Since the clock is free running during configuration, the FD components will latch in the correct initial state value. We must take care, however, to prevent the shift register from actually circularly shifting the waveform data during configuration.

We require the FD components to only clock data from the RPFDs for as long as we are configuring the FPGA. As soon as configuration ends, however, we must simultaneously change the input sources of each FD so that the cyclic shift structure is established. To do this, we place a multiplexor at the input of each FD component and use the select input of the multiplexor to source input data from either the associated RPFD or the output of the previous FD. Tying the select inputs of all control multiplexors to the same initialisation signal allows us to switch between the initialisation and shift modes. Figure 5.8 shows the two effective modes of the control shift-register.

ConfigOK is a FastMap control signal and indicates when a valid configuration has been loaded into the configuration SRAM of the device. In detail, this signal is asserted when the correct values have been written into the XC6200's device ID register. When the XC6200 is reset, the device ID registers are set to zero and the ConfigOK signal is de-asserted. By convention, the last group of configuration writes to occur in a XC6200 configuration bitstream load the appropriate values into the ID register. We therefore know that the ConfigOK signal should only be asserted when the circuitry contained in the bitstream has been configured

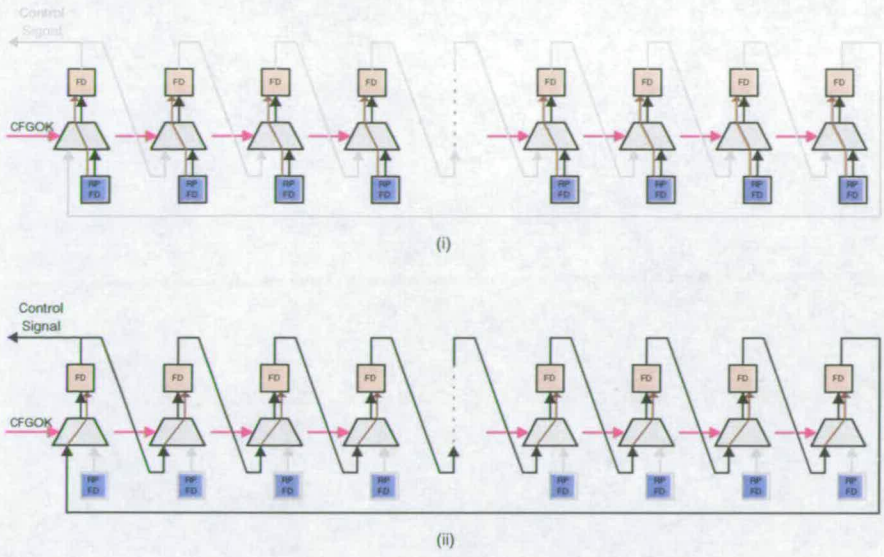

Figure 5.8: The Control Shift-register in action: (i) Initialisation mode; and (ii) Shift Mode.

onto the array. Recalling our discussion of the XC6200 programming interface in Chapter 3, it is possible to access the FastMap control interface from user circuitry implemented within the array. This allows the `ConfigOK` signal to be used as the select net for the shift-register multiplexors and facilitates the self-activation of the control logic. At the end of the configuration process, the `ConfigOK` signal is asserted and the shift-register is then forced out of initialisation mode and into shift mode.

Using shift registers to implement the XC6200 URISC control logic may at first seem wasteful of logic area. A more traditional approach to control logic design is to combine a counter with a decoder circuit. The decoder is hard coded to translate each counter value into the appropriate control signals. Logic minimisation techniques can reduce the area cost of the control decoder. Such an implementation is conceivable for the XC6200 URISC, and could be designed in such a way that it satisfies the self-initialising and self-activating constraints. That is, we would simply use the `ConfigOK` signal to enable the connection from the control counter's output to the control decoder's input.

The shift register approach has one primary advantage for the FPGA implementation: the logic for each control signal is separate and when floorplanning the URISC core, we can place each control-logic signal close to the datapath elements that it influences. This degree of floorplanning flexibility is very useful for reducing routing delays in the URISC circuitry. Furthermore, the shift register design, as we have described it, actually maps well to the underlying cell architecture of the XC6200. The circuit inherently has regular layout and routing and the FD and multiplexor can be mapped into a single cell with the RPFD in an adjacent cell. Furthermore, the same fundamental design is required to ensure the program counter is initialised with the entry point for the system program.

5.2 The Flexible URISC(FURI)

5.2.1 Differentiating FURI and the URISC

The FURI core is a realisation of the abstract architecture we presented in Chapter 4 and is evolved from the URISC implementation we described in the previous section. The primary technical difference between the two implementations is that the memory interface of the FURI core is extended to include access to the host XC6200's FastMap interface. This is in contrast to the earlier, basic URISC implementation where data is only moved between SRAM locations or from SRAM to the PC. By allowing the FURI core to control the underlying configuration SRAM of the host FPGA, we integrate the host FPGA's memory map into the memory map of the FURI core. This effectively creates a self-modifying system. Programs that execute on the FURI core can exploit the FastMap interface to configure, communicate, and compute with the "system bus" SLUs. The fundamental programming model of the URISC is not changed the integration of the FastMap interface: the move is still performs the same basic function.

5.2.2 FURI Implementation Details and Challenges

Figure 5.9 shows the datapath of the FURI core. The main thing to note from this diagram is that there is little additional datapath logic involved in integrating the FastMap interface with the URISC core. This is mainly because there is only a small semantic gap that we must bridge between the two kinds of memory interface used in the URISC core. That given, there are still challenging technical issues that must be addressed in implementing the datapath logic that is required.

5.2.2.1 The FURI Datapath

There are three main changes to the IEU datapath: first is the addition of the FastMap interface ports themselves; the second is a subtle modification to the PC incrementor; and third is the addition of a multiplexor to manage the input source of the IEU incoming data register. The form and, where necessary, motivation of each change is discussed below.

The FastMap interface consists of three main port types: an address port; the input and output data ports; and control signal ports. The format of the data and control ports is essentially fixed, and independent of the exact model of XC6200 being used. The width of the FastMap address port varies depending on the size of the device being used. For example, the address bus of the XC6216 is 16 bits wide whilst the XC6264 requires an 18-bit address. The raw width of the data bus is 32 bits, although the interface can be programmed to react as if it was actually a 16-bit or 8-bit interface, disregarding the inactive interface bits. This variability in the effective data bus width has an impact on the way that we interpret XC6200 addresses. In 32-bit mode, addresses align to word boundaries and the two least-significant address bits are ignored. In 16-bit mode, we address at nibble boundaries, ignoring the least significant bit, and in 8-bit mode all address bits are significant since each byte is individually addressable.

The second alteration to the datapath concerns the PC incrementor. In the

earlier URISC implementation, the addressing of the memory interface operated on the granularity of words. In the FastMap interface, however, addressing is at the granularity of bytes. The FURI memory map, therefore, basically comprises two address spaces which operate at different granularities. Supporting both interfaces translates to the requirement that the PC incrementor must alter the increment value to either 1 or 4, depending on the type of address the PC currently contains. This raises an interesting design issue: how it is possible to differentiate an external memory address and a FastMap address. We could argue that FastMap addresses should never be loaded into the PC, and that instruction sequences are always loaded from memory. It is clear in the later sections of this chapter that this is not a sustainable argument since, in some situations, is imperative that the PC and its incrementor be able to handle FastMap addresses.

A simple convention is adopted to differentiate FastMap addresses from standard memory addresses. A FURI address contains an additional bit in the most significant bit position which indicates whether the address is in the FastMap segment of the memory map, or the standard memory segment. This approach is useful because of its low decoding overhead: we do not need to place complex decoding logic within the datapath to determine the type of an address. For example, the PC incrementor simply examines the most-significant address bit on its input and tailors the increment value appropriately. Furthermore, the changes to the actual incrementor circuitry are small since there are only two potential increment values to select from.

Adding the FastMap interface ports to the datapath means there are now two potential data sources for the IEU temporary incoming register. The final addition to the datapath, therefore, is the introduction of a multiplexor to select whether data arriving on the FastMap or the standard memory interface should be latched. The incoming data multiplexor is mentioned as a separate entity from the FastMap ports because of its control requirement. The selection control

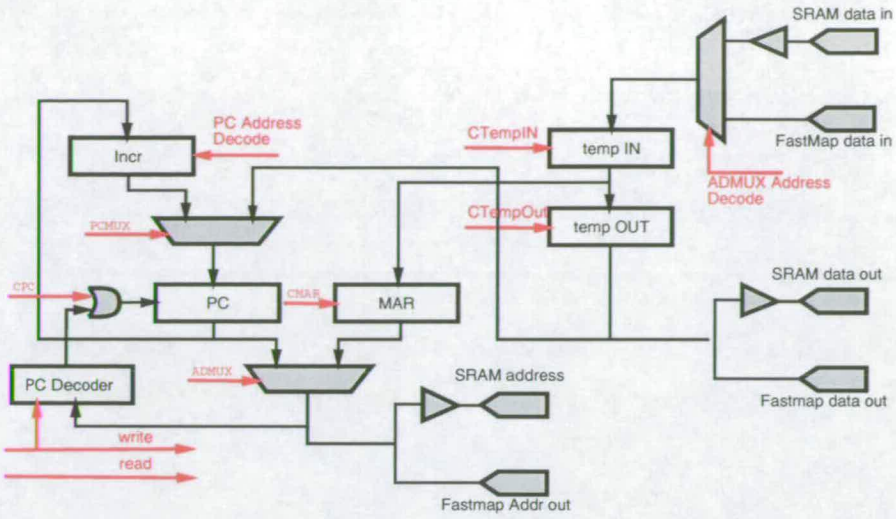

Figure 5.9: Datapath of the Flexible URISC.

for this multiplexor actually comes from an address decoding of the value being asserted by the address multiplexor, ADMUX. In the first three microcycles, the upper bit of this address determines which of the two potential data sources is the correct one.

5.2.2.2 Accessing the FastMap Interface from user circuitry

One of the unique features of the XC6200 series, and one which underpins the FURI implementation, is that circuitry configured on the array can have both read and write access to its control interface. There are technical difficulties in achieving this state, however. In Chapter 3 we gave a detailed discussion of the form and semantics of the XC6200 control interface. We have already begun to use some of the control interface features through our exploitation of the `ConfigOK` signal in Section 5.1.3.2.

Each signal of the FastMap interface is associated with a particular IOB in the array. A given signal can made available within the array with the appropriate configuration of its associated IOB. The standard design flow for a XC6200 device includes a special type of design symbol called a `cbuf`, or control buffer, for exactly this purpose. The circuit designer may instantiate input or output `cbufs` within

their circuit design. The input and output ports of each `cbuf` can be sampled or driven by any user signals which are subsequently attached to it. To access the `ConfigOK` signal used in the previous section, for example, we instantiate the `cbuf_out` design primitive and use the appropriate mechanisms in the design environment to identify that control buffer as the signal source for `ConfigOK`[5]. When the design description is processed by the XC6200 design environment, the resulting bitstream will configure the IOB associated with the `ConfigOK` signal such that one of the IOB outputs is a source for `ConfigOK` within the user design.

`ConfigOK` is an example of a benign control signal. By benign, we mean that instantiating a `cbuf` and configuring an IOB with the resulting bitstream for that `cbuf` will not affect the external operation of the XC6200. However, this is not the case for all control signals. The heart of the problem is that the FastMap interface, as it is implemented on the XC6200, is single-ported. We may configure IOBs to provide access to control signals on a signal by signal basis. In doing so, however, the FastMap control logic no longer responds to any control signals being asserted through device pins: if we configure a signal for internal access, it is no longer available externally. This appears, initially, to be an appropriate design decision. We wish to avoid physical signal contention that could potentially damage the device and it is therefore fair to assert that only one signal source should be directly, physically wired to the signal input for the FastMap control logic.

For control signals like `ConfigOK` the external isolation problem does not apply. The `ConfigOK` signal is actually classed as an *internal* control signal: it is only available for use inside the cell array through a CBUF. Simply configuring access to internal control signals does not pose an immediate threat to the general operation of the hardware system. The act of configuration does not change the

[5]VHDL was used as the description language in the URISC and FURI implementations. VHDL 'attributes' are the language mechanism used to associate each `cbuf` instance with a particular control signal.

external device interface in this case. Our main problem arises when we try to assume control of signals such as the FastMap address and data inputs. Here, simply configuring the array to allow access to the control signal is enough to alter the device's external interface. Within the space of one such configuration, we can alter the external interface enough to make the device inaccessible. For example, the FURI core requires complete write access to the FastMap address port. This is done by configuring the appropriate address signal `cbuf` instances one at a time. However, the section of configuration address space represented by each address line we assume control of becomes unavailable in any future configuration.

We cannot avoid the problem by seeking a configuration sequence for FastMap `cbufs` that would leave enough of the FastMap interface available externally at any one point to be able to complete the entire configuration. Configuring `cbufs` for the incoming data port prevents us from completing configurations for the address port and vice versa. We can survive for a short time with limited access to the configuration address space. However, configuring the FastMap chip-select signal for internal access would make the device *immediately* inaccessible since all subsequent external FastMap transactions involve strobing of the external chip-select pin.

An alternative approach involves using the XC6200's wildcarding facilities to simultaneously configure all the relevant IOBs with the desired `cbuf` configuration. This is theoretically a more elegant solution since it would provide a single point in time where device control switches between the external and internal FastMap ports. However, this approach has three primary requirements which we cannot completely satisfy: first, the target IOBs must have a suitably regular geometric arrangement that can be encoded by column and row wildcards; secondly, we must be able to pack the critical configuration data into a single write over the FastMap interface; and, finally, it must be the same configuration

data that is applied to each target IOB. We find an immediate problem with the first requirement since the IOBs used for the address, data, chip select and read-write signals are spread across the East, South, and West edges of the array. We must therefore broadcast configuration data over three of the four device edges to achieve our task. If we couple this with physical limitations[6] of the wildcarding facilities, then the approach quickly becomes intractable.

5.2.2.3 Bootstrapping with the Serial Interface

In short, we know from the previous section that we cannot use the external FastMap ports to apply the configurations which give complete control of the FastMap signals to internal circuitry. However, the underlying assumption up to this point is that we *must* use the FastMap interface to perform the inital configuration of the FURI core's FastMap logic. The alternative which is successfully employed in the FURI implementation bypasses the FastMap interface altogether during the bootstrapping phase of the FURI core cicruitry and, instead, applies the initial configuration through the XC6200's serial interface. However, once the initial boot configuration of the core has been successfully configured onto the target XC6200, the serial interface is discarded. From that point, all subsequent interactions with the host FPGA occur through the FURI core's configured interface to the host's FastMap port.

Serial interfaces generally operate in both 'slave' and 'master' modes: in slave mode, the device responds passively to control signals asserted from an external source; a device in master mode actively asserts its own control signals to source configuration data from a passive configuration store. The serial interface of the XC6200 is similar to the serial interfaces in most mainstream commercial FPGA architectures and, specifically, consists of the six dedicated device pins which are

[6] details of the wildcarding mechanism in XC6200 literature[109] state that there is a limit to the number of cells which can be simultaneously written using the mechanism. In each device model, the number of addresses we can simultaneously target is smaller than the number of simultaneous writes that we must apply to complete the switch-over between interface ports.

Signal	Rôle
/serial	Controls whether the XC6200 should enter serial or parallel mode
wait	Causes the XC6200 to stop loading configuration data until the signal is de-asserted.
SEReset	Asserted by the XC6200 to reset the serial data source and prepare it for subsequent access
/SECE	A chip-enable signal asserted by the master FPGA to enable data output in the serial PROM
SEClk	The clock signal to which serial transmissions are synchronised
SEData	Data being received from the PROM for consumption by the serial interface.

Table 5.3: Control signals used in the XC6200 serial interface

listed and explained in Table 5.3. Data are shifted into the XC6200 over the serial interface and is then passed to the FastMap control logic. The FastMap operations that can be done through the parallel interface can also be done serially.

Whilst it is true that the control logic design presented earlier does not require any external control signal to kick-start the URISC core, it is not clear how the URISC bitstream arrives at the FPGA in the first place. If we rely on the services of an external agent to actively configure the XC6200 then the implementation is not truly autonomous. Using the serial interface to communicate the initial design actually addresses both issues: it allows us to configure each IOB with an appropriate `cbuf` to facilitate internal access to the desired FastMap signals; and, second, it serves as a low-level physical mechanism for making the FURI core autonomous. When the XC6200 is powering up or is reset, the signals asserted on the `/serial`[7] and `wait` pins define whether the device enters master serial, slave serial, or parallel mode. The device context for the autonomous FURI implementation is shown in Figure 5.10. The FURI bitstream is held in a serial PROM which is hardwired to the XC6200 serial interface. Physically tying the `/serial` and `wait` pins to ground initially locks the device into master serial

[7] signal names with a preceding / are active-low.

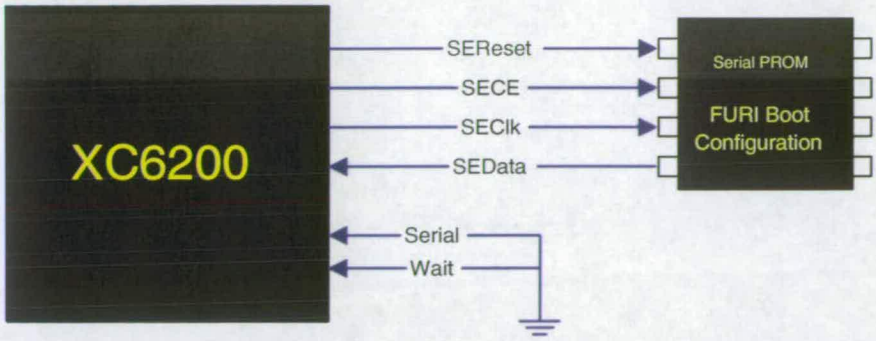

Figure 5.10: Autonomous FURI system using XC6200 master serial configuration

mode. After the XC6200 completes its internal power-up sequence, it actively reads the FURI bitstream held in the serial PROM. In this organisation, no external control source is needed to manage the boot-configuration of a FURI core.

5.2.3 FURI Control Logic

Fundamentally, the same control logic design is used for the FastMap control signals as is used in the earlier basic URISC implementation. However, it is necessary to scale the circuitry slightly to accommodate changes in the control timing diagram that arise from the inclusion of the FastMap interface in the IEU memory map.

5.2.3.1 FastMap Timing

Figure 5.11 contains the timing diagrams which specify the series of events that the IEU control logic must trigger to perform a FastMap read and write to the XC6200 configuration memory. The diagrams show examples of both basic and extended transactions over the FastMap interface. The FastMap interface operates synchronously with respect to the global clock signal, `GClk`. The dashed lines in each timing diagram show the points where the interface signals are sampled by the XC6200's internal control logic. This always occurs at the rising edge of the `GClk` signal. In a basic read or write cycle, `/cs` is first sampled low and

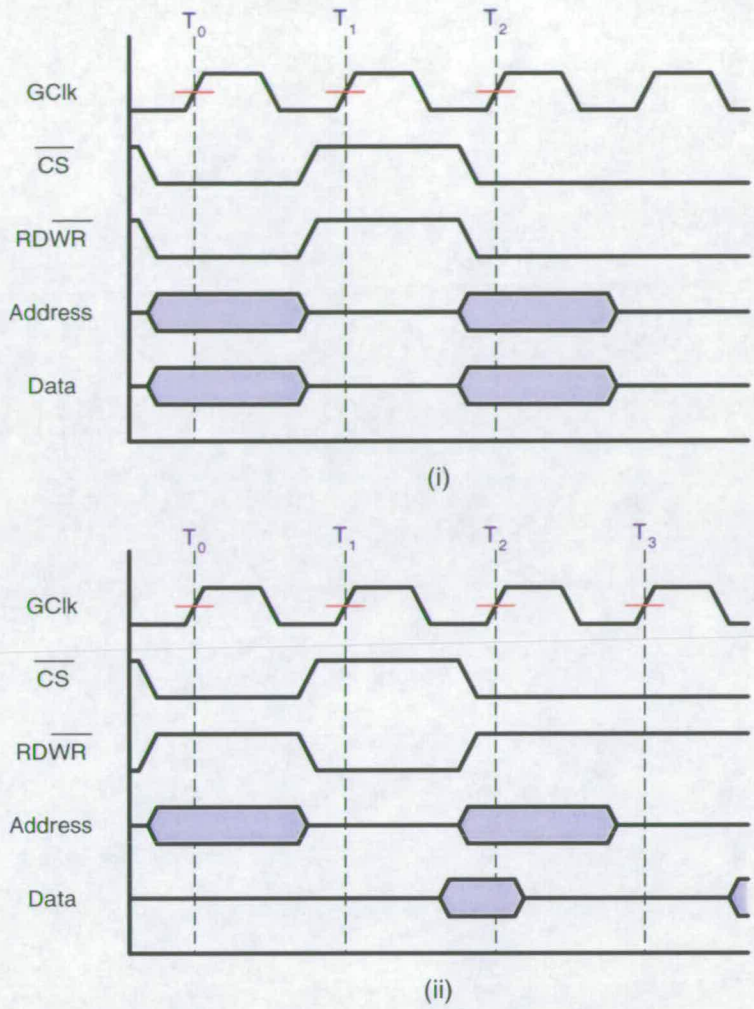

Figure 5.11: FastMap Interface Timing Diagrams: (i) Configuration SRAM Write; (ii) Configuration SRAM Read

then sampled high before another cycle can start. If a longer sequence of reads or writes is to be undertaken, a basic cycle can become an extended cycle by returning `/cs` to low immediately after it has been sampled high, at the end of the basic cycle. The FastMap interface will continue to process inputs until the `/cs` signal becomes high again, signalling the end of the extended cycle.

The timing requirements for a FastMap state access differ slightly from a configuration SRAM access. A state read takes longer than a configuration SRAM read, although writes have the same cycle time. The FURI implementations in this thesis have moderate system clock speeds of up to 32MHz and the longer

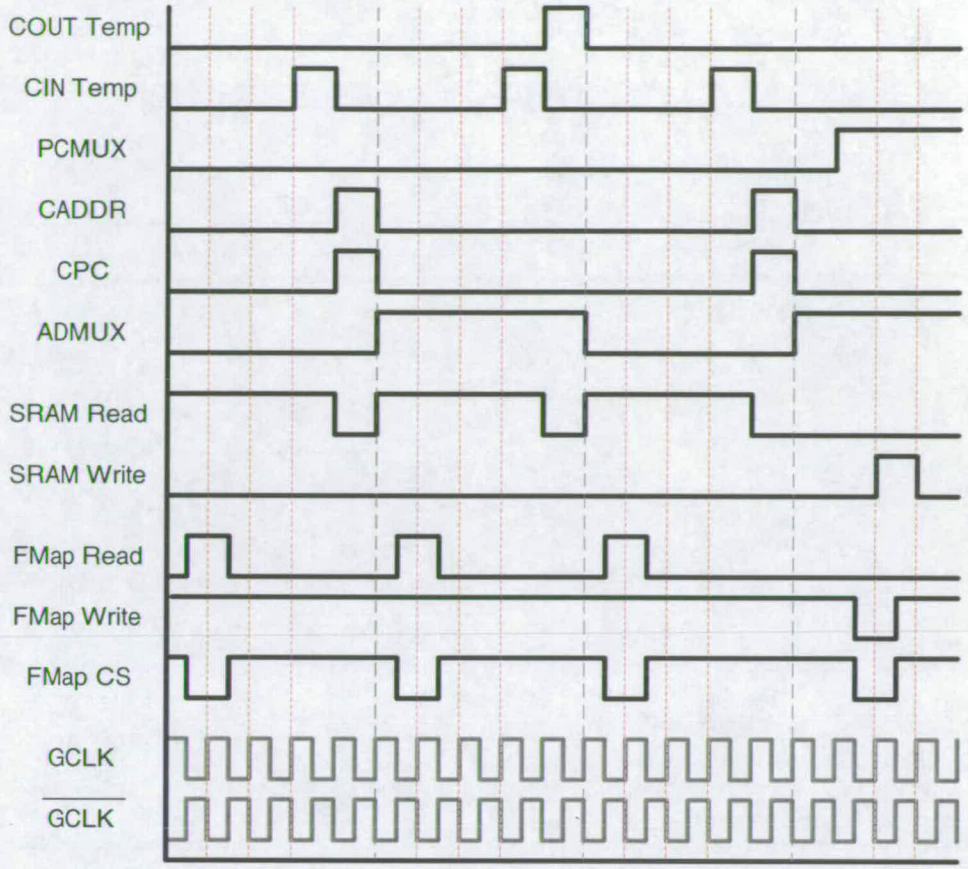

Figure 5.12: FURI Control Timing with integrated FastMap Read and Write support

state cycle time does not pose a particular problem. We should note that, at this stage of the thesis, the intent is not to explore circuit level performance enhancements to the FURI core. The main challenge here is implementing the standard architecture itself. However, if the system clock period was sufficiently high, it would be necessary to have separate control logic to manage state accesses.

A FastMap transaction may be involved in at least two of the four FURI microcycles. For example, our general aim is to exploit the FastMap interface within the FURI core as a means of customising which components are present on the URISC system bus, and also interact with those components. In terms of FURI accesses to the FastMap interface, this translates to being capable of reading and writing both configuration and state data during the second and

fourth FURI microcycles. The actual source and destination addresses of each move instruction are still fetched from external memory and we do not consider program code which may be embedded within the cell array itself. Section 6.3.2, however, describes a situation where it is imperative that the FURI core can execute such 'internal' code. We therefore support FastMap transactions in all four microcycles.

Although the timing diagrams in Figure 5.11 appear fairly simple, there are some challenges involved in integrating them with the underlying URISC control. The first challenge is to meet the hard timing requirements for the correct sampling of the `/cs` signal. If the `/cs` signal is sampled low at T_0 then it *must* be sampled high exactly one `GClk` cycle later, at T_1. If this does not happen, the transaction will not complete. In the basic cycle, other signals in the timing diagram need only be sampled correctly at T_0.

Furthermore, the timing diagrams in Figure 5.11 show that the signals that driven onto the FastMap interface are slightly out of phase with the main `GClk` signal. This is because the FastMap control signals each have minimum setup time constraints. We must guarantee that these constraints will be met before the signals are sampled at the rising edge of `GClk`. To facilitate this, the implementation of the FURI core presented here synthesises a FastMap control clock that is 180^o out of phase from the main `GClk`. This is done by routing the `GClk` signal through an inverter and, from there, directly to the input of one of the XC6200's low-skew global nets. A low-skew global net is appropriate for use in this instance since the FastMap control logic, like the general URISC control logic, may be spatially separated across the cell array. Using standard routing resources would undoubtedly result in subtle signal skews which could then interfere with the correct timing behaviour of the control logic.

Figure 5.12 contains a modified timing diagram showing the FURI control timing, including the FastMap signals applied during each microcycle. One of

the first things to note about this diagram is that an instruction cycle now takes 19 clock cycles. The first three instruction microcycles are extended by one clock cycle. This is to accommodate the three clock cycles required for a FastMap read transaction. In the original URISC timing, only the first two cycles in a microcycle were required to setup an SRAM read transaction: in the first cycle, the address is asserted on the bus and on the second cycle the appropriate control signal is asserted[8].

At first glance, the three clock cycle FastMap transaction does not appear problematic since the incoming data register will also latch its input on the rising edge of the third clock period. However, it the phase difference between the standard URISC control signals and the FastMap controls that complicates matters and provides the motivation for the extra clock period. In short, the extra `GClk` cycle allows us to meet the setup timing constraints for `/cs` signal by giving `/cs` the time to do a complete transition from high to low then back to high. If `/cs` is sampled low at the second `GClk` and high at the third, the FastMap data will still not be valid until the rising edge of the fourth `GClk`. This is too late for standard URISC timing since the incoming temporary data register latches its input on the rising edge of the third `GClk`.

An alternative solution involves initialising the `/cs` signal to start at low and rise to high half way through the first `GClk` period. Correspondingly, for future instruction cycles `/cs` must then begin its transition to low at the end of the fourth microcycle. We rely on the mechanics of the self-initialising control registers to ensure that `/cs` is initialised to the correct value. The net effect, however, is that the FastMap transaction begins one cycle earlier and falls back into alignment with the original URISC timing. Provided we guarantee the validity of the appropriate address as the first `GClk` rises, the first three microcycles could be

[8]it is acknowledged that this is also not the most efficient timing and that, when this thesis was written in 2000, it is commonplace for SRAM interfaces to allow simultaneous assertion of address and control signals.

again shortened to four clock periods. The FURI implementation used in this thesis does not use this timing approach, although it is presented as a potential design enhancement that would increase the core performance.

The fourth instruction microcycle has a simpler implementation since a basic FastMap write has only two significant `GClk` sampling points: it is not necessary to wait for any result on the FastMap data port. Again we allocate the first clock period in the microcycle to setting up the address and, in this case, data buses. In the second clock period, the PC multiplexor is set to allow the program counter to be overwritten whilst the `/cs` signal simultaneously makes its high-to-low transition within the setup time constraint. By the time we reach the rising edge of the third `GClk`, `/cs` has settled low and the FastMap write can begin. It is also during the third `GClk` period that `/cs` will make its rise back to high ensuring the minimum setup period will have been met when the signal is sampled at the rising edge of the fourth clock.

5.2.3.2 Implementation of the FastMap Control Circuitry

In the timing diagrams presented so far, the read and write events have been separated into two different control signals. This has been the case for both the FastMap and basic memory control timings. We should note that, in reality, both signals are physically represented by a single "read-not-write" control. This is evident in the timing diagrams for the FastMap interface and the FURI implementation of the FastMap: the first diagram has only a single `rd/wr` control whereas the second has two separate signals. In the original URISC timing diagram, the active state of a device was derived from the current assertion on the read and write controls. Since the two signals are operationally mutually exclusive, if neither is driven then we can deduce that the target device is inactive. Conversely, the target device is activated by an assertion of either control input.

In reality, both memory interfaces in the FURI core use an explicit chip-select signal to identify when the interface should be active. This allows a single

read-write signal to identify which action should be performed when the device is active. Originally, the read and write controls were presented separately to provide a slightly more intuitive indication of what a particular interface was doing in a given clock period. For example, a single pulse on the read control at a given clock period clearly identifies that a read is occurring, when it is occurring, and how long it lasts. Furthermore, we need not explicitly cross-referencing the state of a chip-enable signal to identify whether the action is actually happening, or whether it is ignored.

In the first design iterations of the FURI core, the FastMap read and write signals were indeed instantiated as two distinct control logic shift registers. This was motivated by a potential area saving based on the observation that, when taken separately, the two signals were simply repetitions of bit sequences of length 5 at most. Two smaller shift registers of length 4 and 5 respectively were designed to capture the repeating patterns of the FastMap read and write signals. A simple multiplexor was used to select which of the two signals should be driven onto the actual FastMap `rd/wr` input. The FastMap write control shift-register should drive the read-write port during microcycle four with the read control register driving at all other times. Since the `PCMUX` signal is only ever active during the fourth microcycle, it is reused as the select input to the FastMap read/write multiplexor[9].

A further motivation for reducing the size of the FastMap control logic was to attempt to balance the routability of the FURI core with the timing requirements of the FastMap control signals. A floorplan of the FURI core that surrounds the FastMap control signals is shown in Figure 5.13. The main point to draw from this diagram is that there is a large amount of datapath logic surrounding the FastMap control signal IOBs. Additionally, the access points for the SRAM

[9] Using the `PCMUX` signal in this manner is not an optimal solution to this problem as it adds extra semantics to the signal. However, it is also arguable that the `PCMUX` is already asserted for longer than it need be: in the original URISC literature, the PC multiplexor is enabled for all of microcycle four when it could be restricted to match the timing of the `CPC` signal.

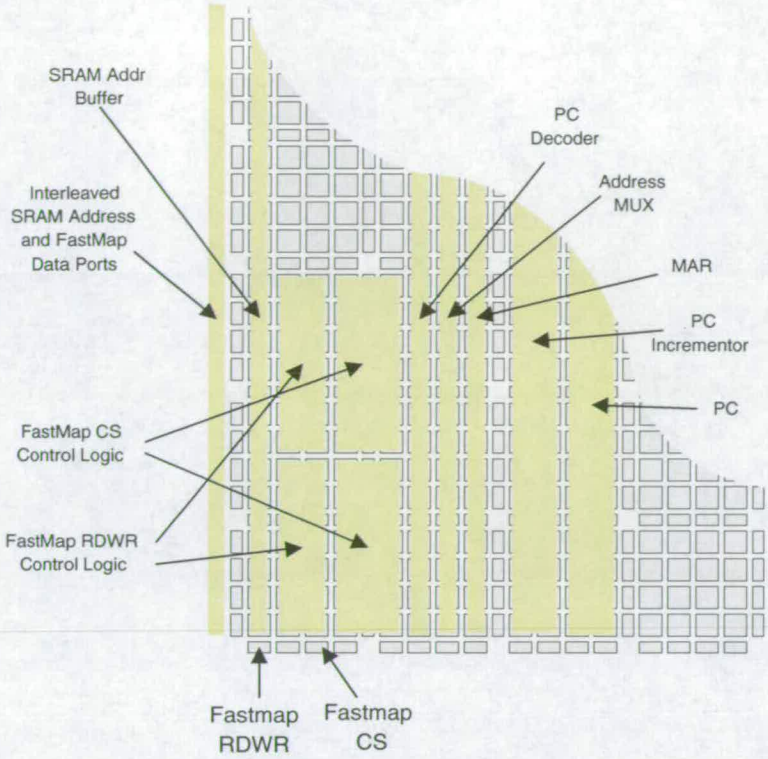

Figure 5.13: Floorplan of FURI core around the FastMap control Ports

address and FastMap data port are on the west edge of the cell array. This increases the routing congestion in this part of the cell array further since the nature of these signals meant they consumed significant amounts of the upper levels of the XC6200 routing hierarchy. Essentially, the `/cs` and `rd/wr` control logic must be placed close to the IOBs providing those FastMap ports. The `/cs` control logic was implemented similarly.

In the final design iteration, shift register logic for read and write control was removed completely and control for the `/cs` signal was implemented as a single 19-bit shift register. Our main motivation for taking this approach is to increase the routability of the design in the congested region around the FastMap control signal ports. In the strictest technical terms, the resulting control waveform does not match the FastMap timing since the FastMap `read` and `write` signals are now asserted for entire URISC microcycles. However, there is enough tolerance

in the specification of the interface to maintain correct operation. The `/cs` logic is clocked from the inverted `GClk` signal to facilitate the phase shift between the FastMap and normal core signals.

5.2.4 Debugging the FURI Core Circuitry

5.2.4.1 The Effects of Device Isolation

One of the greatest strengths of the XC6200 architecture is the relative ease that circuitry can be interrogated and interacted with using the FastMap interface from the outside environment. However, we lose this valuable debugging interface and the XC6200 effectively becomes isolated from the outside world when we give control of the FastMap interface to circuitry within the cell array. This complicates the development of circuitry and programs for the FURI system since the FastMap interface cannot be used to monitor the progress of the circuitry and core as the program executes. Developing and testing the FURI core circuitry itself is challenging in this environment. Since the FastMap interface becomes inactive during serial configuration it is often not immediately clear if a failure is caused during configuration or because of an error in the executing FURI program.

The isolation of the XC6200 from the external system has a significant impact at different design levels in the FURI system. In this section, our primary concern is describing the particular approach taken during the implementation and debugging of the FURI circuitry. Immediately following this section, however, is a larger presentation of the FURI design and programming environment. This includes details of the tools used to develop the FURI core and, in particular, a tool to help compensate for the isolation of the FURI core when designing circuitry off-line.

A lack of access to the FastMap interface means it is necessary to resort to *physically* debugging the FURI core circuitry. That is, be of analytical instrumen-

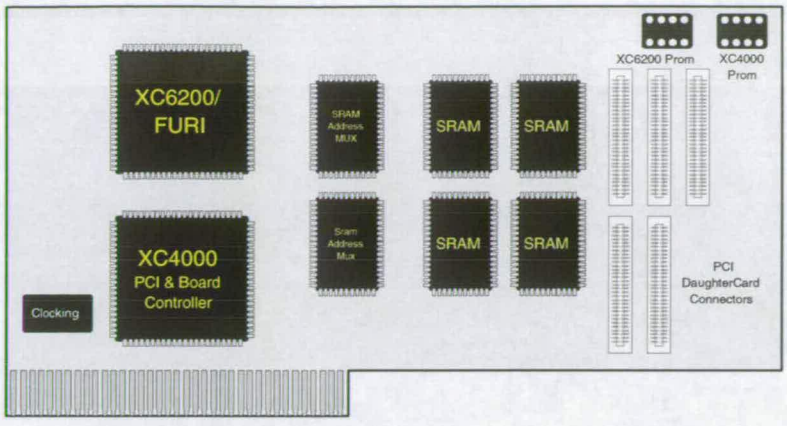

Figure 5.14: The VCC Hotworks Development Card

tation such as logic analysers and oscilliscopes to monitor FURI circuitry state via the signals driven on the XC6200's device pins. Generally speaking, this is a much poorer debugging interface than the FastMap. We are limited to passively observing the device outputs and, if we wish to observe an internal core signal, it must first be routed to a device pin that is accessible to the logic analyser probes. To complicate matters further, there are only a limited number of device pins that we can use to output the state of internal signals. The analogue nature of physical debugging does serve as an advantage. The physical traces we collect are actual samples of the circuitry outputs taken in real time. This can reveal physical timing glitches and hazards that are otherwise not evident through the synchronous FastMap. Since we are using multiple clocks of different phases, being able to physically observe any such behaviours in the core circuitry is valuable.

5.2.4.2 The FURI Development Platform

To give an adequate description of the hardware testing platform, some general details of the hardware platform used for FURI development are appropriate at this point. The URISC and FURI implementations described so far have both been targeted at a PCI prototyping card that is compliant with the Xilinx

XC6200 Development System architecture[84, 30]. In particular, they have both been implemented on a VCC Hotworks development card. The Hotworks/XC6200 board architecture is shown in Figure 5.14. For our current aim, the main thing to notice from this architecture is the existence of a series of PCI daughtercard connectors. These connectors are physically routed to the device pins of the XC6200 and allow a daughtercard to physically interact with the XC6200. Figure 5.15 shows the VCC Hotworks prototyping daughtercard which maps selected pins from the XC6200 to a collection of wire-wrap header pins. Logic analyser probes attached to these pins can observe data being transferred between the XC6200 and the onboard SRAM and also from the serial PROM to the XC6200 serial interface. The `DA` and `DB` ports are mapped to the IOBs on the North edge of the FPGA and form part of a console (CON) port. This port is also available as a readable device register implemented by the XC4000 board controller. Software executing on the host processor can read values driven onto the CON port IOBs and, theoretically, we could use this register to inspect the internal state of the device. However, since we do not have write access to the XC6200, it is not possible to use single-step software clocking: the FURI core must execute at full clock speed. A logic analyser is more appropriate for capturing such free-running signal traces. Fortunately, the signals from user logic driven to the CON port IOBs can be physically sampled at the prototyping card's corresponding headers.

5.2.4.3 The FURI Design-Debug Test Cycle

The design-debug cycle for testing the FURI core circuitry is shown in figure 5.16. The diagram shows some of the tools and auxiliary design tasks that are involved at different stages. The first two stages of the cycle are simply design capture, compilation, and mapping. With the FastMap interface, it is rarely necessary to instantiate additional logic to support debugging alone[10]. Here, however, we must configure and explicitly route internal user signals to IOBs so that we can sample

[10] A form of software controlled clocking is the exception to this rule.

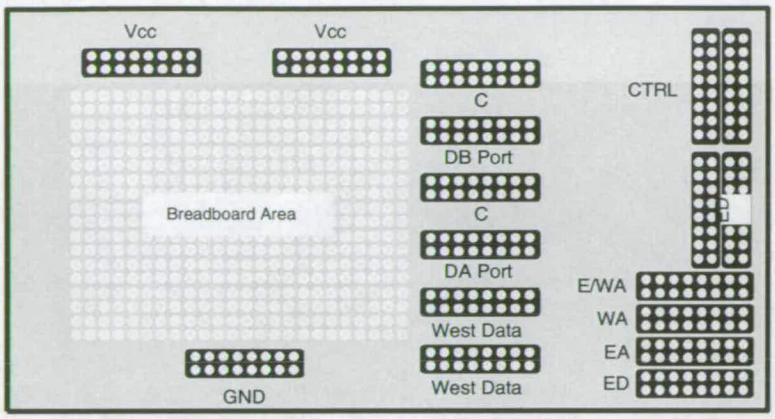

Figure 5.15: The VCC Hotworks Prototyping Daughtercard

them with logic analyser probes attached to the daughtercard. The system clock speed is low enough in this instance that there are no adverse effects on circuit timing. Expressing this additional debugging logic in VHDL is not difficult but makes the place and route task even more demanding. Indeed, routing congestion means it is sometimes necessary to limit the number of signals routed to the IOBs.

Stages three and four are an artifact of our use of the serial programming interface on the Hotworks board to load the FURI bitstream. We must program each new bitstream onto a PROM or, preferably, EEPROM and then physically install the new EEPROM in the appropriate board socket. This is, generally, one of the most cumbersome ways to achieve our serial configuration goal. More elegant solutions could have involved in-system re-programming of the EEPROM or wiring the serial interface directly to a software accessable register in the XC4000 board controller. Unfortunately neither of these were supported by the development platform.

In the next stage, the testing equipment must be physically reconfigured to match, for example, any changes in the signals mapped to the debugging interface. This includes recalibrating and reprogramming the equipment, and also updating the probe attachment configuration to match any changes to the debugging signals driven to the prototyping card. Once this is done, we can capture a signal trace

of the FURI core in operation. In our earlier discussions regarding the XC6200 serial interface, we considered both master and slave serial configuration and advocated the use of master configuration to facilitate an autonomous FURI. In the Hotworks platform, the XC6200 has a slave relationship to the serial PROM and relies on the services of the board controller to initiate the serial download. Whilst the current implementation does not quite achieve full autonomy, this is an attribute of the development platform used and not a symptom of a fundamental inability to achieve full autonomy.

A simple board console program, QPCItest, allows us to trigger the initial serial configuration from the host system and is described in more detail in a later section. An auxiliary design task at this stage is to create small fragments of FURI code that will exercise the core feature we wish to test. This would include, for example, hand coding a small sequence of instructions that repeatedly cause a FastMap transaction. The board console also facilitates loading this instruction sequence into the appropriate region of onboard memory. The serial configuration sequence is then initiated and we capture traces of the internal FURI signals SRAM bus transactions as they occur in real time. The final cycle stage is dedicated to analysing these waveforms and traces to identify the existence any errors and deduce their cause.

5.2.4.4 Reducing the number of debug iterations

Each design change to the FURI core involved an iteration of this long debug cycle. The incremental approach used in the development programme, however, helped to reduce the number of FURI debug iterations. For example, since the FURI core implementation was evolved directly from the simpler URISC implementation which does not require the isolation of the XC6200 to operate. We could, therefore, debug the URISC implementation relatively rapidly using the facilities of the FastMap interface and single-cycle software clocking. The net effect of this is that it provided a strong foundation and helped to validate the

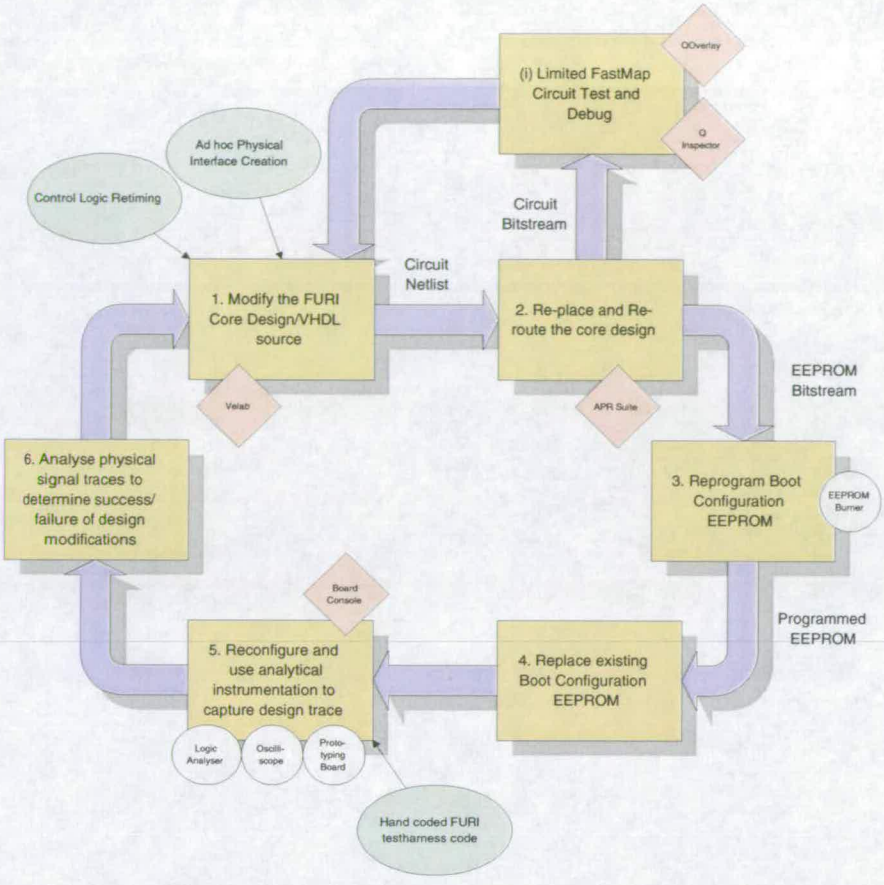

Figure 5.16: FURI Core Hardware Debugging Cycle

FURI core circuitry.

The design-debug cycle in Figure 5.16 shows an limited debugging stage adjacent to the placement and routing of the core design and feeding directly back to the first cycle. Here, we avoid configuring the internal FastMap interface and interact with a limited or constrained version of the FURI core circuitry. For example, a circuit simulation of the FastMap interface was initially designed and attached to the internal address, data, and control paths of the FURI core. The simulated internal interface was designed to respond as closely as possible to the real FastMap interface. Circuitry debugging tools such as QInspector, which is described in the following section, could then be used over the FastMap interface to monitor the FURI core as it interacted with the simulated interface. In the early stages of the FURI core implementation, this path dominated the design-

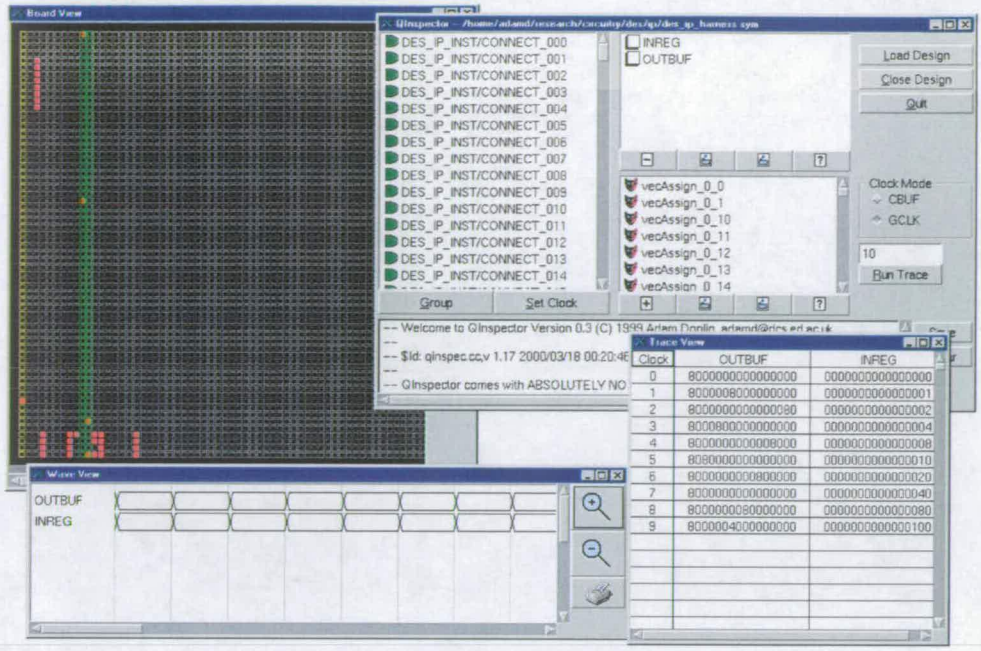

Figure 5.17: `qInspector` Design Views

debug cycle. We only pursued a full debug cycle after the core reached a suitable level of functionality.

5.2.4.5 QInspector

`qInspector` is a Linux tool that facilitates interactive, visual validation of the functionality of XC6200 SLU circuitry. The tool offers three design views: an interactive board view; and both waveform and trace views of selected symbols within the SLU circuitry. Figure 5.17 shows a snapshot of these views as the main debugging environment used in the development of both the FURI core circuitry and FURI-compatible SLUs. The tool uses the XC6200's FastMap interface to determine the state of the target SLU. A rich scripting language is implemented to allow the SLU designer to apply sequences of test vectors, gather the trace results, and access most of the user interface control functionality.

5.3 Summary

This chapter presented the design and implementation of the Flexible Ultimate RISC, an evolution of the Ultimate RISC. We presented a detailed discussion on the challenges of implementing a self-modifying microarchitecture on the target FPGA architecture. From here we then considered some features of the development and debugging process and environment of the FURI core.

Chapter 6

The FURI Programming and Runtime Environment

In this chapter we present details of a programming and runtime environment for the FURI core described in the previous chapter. The chapter has two main components:

- First, we present details of the design flow and an associated toolset for programming the FURI core. This discussion pays particular attention to the merits and complications associated with different approaches to loading SLU bitstreams.
- Second, we introduce the FURI executive as a basic, multitasking runtime environment for the FURI core. During this discussion we describe a base protocol for interacting with the executive and consider issues such as FURI code embedded within the host FPGA itself.

6.1 Programming the FURI Core

So far, we have presented a comprehensive discussion of the design, operation and implementation of the FURI core. What we have yet to present, however, is a programming environment and associated tools that will allow us to exploit the FURI core. Figure 6.1 shows the design flow that has been developed to facilitate FURI system construction. Before we can explore this in detail, however, it is

worthwhile clarifying the definition of a FURI program.

6.1.1 What is a FURI program?

In a fundamental sense, a FURI program is simply the sequence of move instructions that is executed by the FURI core. However, when taken literally, the raw sequence of moves rarely amounts to any form of complex calculation. As a transport-triggered architecture, FURI depends on the availability of the appropriate system bus SLUs for use as transport targets. A FURI *application*, therefore, comprises a collection of SLUs that are used as targets for the data transports effected by the execution of the move instruction sequence on the FURI core. From this point of view, it is clearer that the design flow presented in Figure 6.1 addresses more than just the construction of FURI programs: it is a framework of tools and libraries used to create FURI applications.

6.1.2 The FURI Design Flow

We can broadly partition the design flow into two sections: the right side of the flow addresses the construction and execution of FURI programs whilst the flow to the left considers the construction of SLU circuitry. In between these two sections is a central flow that provides a bridging mechanism to facilitate the loading and interaction of SLUs within a FURI program.

6.1.2.1 SLU Design Flow

SLUs enter the design flow either as unprocessed 'soft' VHDL descriptions or as 'hard' pre-defined SLU bitstreams in circuit libraries. The VHDL side of the design flow uses two of the XC6200 standard design tools to compile SLU descriptions to bitstreams: Velab compiles the structural VHDL description of an SLU into an EDIF netlist and the XC6200 place and route software maps the netlist to the FPGA architecture and generates an appropriate bitstream[1]. Essentially this

[1] XC6200 bitstreams are commonly referred to as CAL files where the term CAL is an historic reference to the Algotronix CAL which preceded the XC6200.

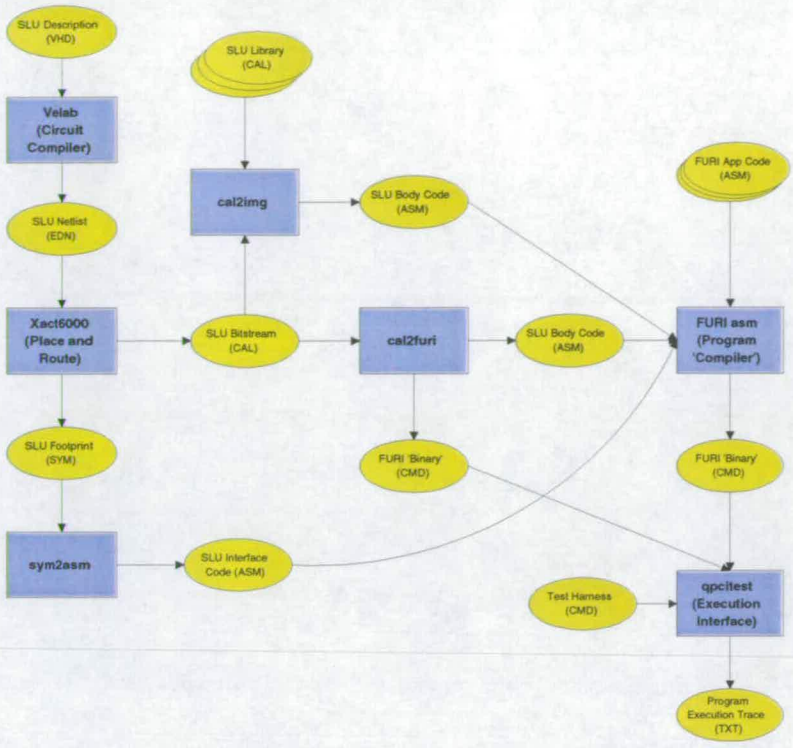

Figure 6.1: The FURI Design Flow

is the same circuitry design flow used for the FURI core circuitry. Rather than outputting a formatted data file for use with an EEPROM programmer, however, the place and route tools produce a raw bitstream.

Once the SLU has been rendered into a bitstream, the intermediate tools are used to convert it into a form that we can use within a FURI program. Two of these tools, `cal2img` and `cal2furi` convert the bitstream into a 'loadable' form. That is, we convert the raw bitstream data into an assembly representation of one form or another that can be used later to instantiate the SLU on the cell array. The two tools, discussed below, generate different assembly representations of a circuit bitstream.

Instantiating the SLU involves the services of a circuit loading subroutine. We will describe different forms of circuit loading routines in the later sections of this chapter, but a short, high level overview of the loading process is appropriate before discussing the loader support tools below. The main task of an SLU

loader is to transfer each word of an SLU's bitstream data from the general data memory into the configuration memory of the host XC6200. The exact sequence of actions performed to transfer the bitstream depends on the organisation of the bitstream in data memory. The loader subroutine understands the structured layout of the bitstream data and can transfer each word of the bitstream image to its appropriate destination within the XC6200's configuration memory.

`cal2furi` takes the bitstream data and generates two files. The first file contains a specialised assembly subroutine that, when run, transfers the target SLU bitstream data from a specific, hardcoded locations in data memory to the XC6200's configuration memory. The second file contains the raw bitstream data encoded as a memory image. This memory image is actually a sequence of commands to the execution interface of the execution environment[2]. When each command is applied, it places a single word of the bitstream data at a particular location in board memory. Each location matches a location that the specialised loader subroutine expects to find a single word of the bitstream. `cal2img` converts the SLU bitstream into an assembly datastructure, along with some assembly constants to describe the contents of the structure. A generalised loader subroutine can be passed these constants as operands, allowing it to dynamically instantiate different bitstreams.

With `cal2furi` it is necessary to define, in advance of the main program assembly, a region in program memory where the SLU bitstream can reside. It is also necessary to explicitly 'link' the SLU memory image with the memory image containing the application program itself. Manually arranging the memory floorplan of the FURI application is not sustainable in anything other than the small scale. However, the `cal2furi` mechanism has the advantage of avoiding the control-flow calculations required by a generalised circuit loader programmed

[2]Here, the execution interface is formed by the debugging tool `qPCItest`. This tool has a script-style command interface. One of the features of this interface allows allows the SRAM of the Hotworks board to be populated with data. It is this operation that we exploit to fill the program and data memory of the Hotworks board prior to activating the FURI core itself.

in FURI assembly.

A third tool, `sym2asm`, generates an assembly-level SLU interface description to facilitate interaction with an SLU after it is loaded. The assembly interface does not contain any program code; rather, it comprises a series of literal declarations and constants that define the position of an SLU within the FURI memory map. In particular, the interface specifies the exact memory locations of each SLU input and output along with the map register values that must be applied when the registers are being accessed. Earlier research [13], has shown that deriving this interface information from the raw XC6200 bitstream is non-trivial. The intention here is not to derive this information from the raw bitstream, although it is noted that the approach used in the XC6200 configuration compression technique[49] shows some promise for this. Instead, `sym2asm` exploits a file containing symbolic information about an SLU that is produced as a by-product of the place and route process.

6.1.3 The FURI Assembler

Expressing anything other than the simplest of algorithms using only the FURI core's `move` instruction is very cumbersome. To facilitate the construction and expression of larger FURI programs, a FURI assembler was developed. The FURI assembler operates on sequences of raw move instructions at its core, but is itself flexible and supports the definition of a higher level instruction set interface using instruction *macros*. This is similar to the dynamic assembler used in the programming environment [22] of the DISC project. In both situations, we have a processor microarchitecture capable of using dynamic reconfiguration to facilitate a flexible instruction set. However, the operational characteristics of the FURI core are quite different from the DISC processor. The DISC dynamic assembler focuses on supporting a flexible instruction set through defining different instruction formats. The FURI assembler, on the other hand, has a fixed

basic instruction and format and facilitates a flexible programming instruction set through macros which define the set of basic data transports to SLUs that are required to implement each particular instruction. A complex 'compound' instruction can be defined from a sequence smaller, simpler instructions. Each macro has a header, which defines the instruction operator name and number of operands, and a body comprising the sequence of instructions which implement the compound instruction. When the FURI assembler is processing a particular assembly file, it substitutes each macro instruction with a specialised instance of the macro body. Using this mechanism, it is possible to build a richer assembly programming interface on top of the minimal FURI core instruction set.

Earlier in this chapter, we discussed the lack of URISC addressing modes and how their absence can be overcome with programming conventions. The FURI assembler supports many of the standard features of an assembler, such as declaration of data literals, symbolic references to those literals, and named instruction labels. The assembler also supports or applies programming conventions used to implement the more complex addressing modes over the underlying absolute addressing mode. For example, the assembler supports immediate addressing explicitly through a table of immediately-addressed literals. Every instance of immediate addressing is converted to an absolute address within this data table which is then included in the final binary image of the program. Indirect addressing is supported in two forms. In the first form, an asterisk '*' operator can be applied to dereference any literal operand during assembly, effectively substituting it for its initially defined literal value. The second form is more dynamic and uses the self-modifying code strategy described earlier in this chapter, but requires that the FURI assembler supports the application of a static offset to an instruction label. This allows us to define the point in the instruction stream that is to be modified relative to a particular instruction label. The implementation of the strategy can be captured as a specialised version of an instruction macro.

6.1.4 Kernel Circuitry

The calculations that underpin basic control flow in a FURI program require that we define a set of 'kernel' circuits. In implementation terms, the kernel circuits are a set of SLUs that are loaded onto the host FPGA after the FURI core circuitry has been bootstrapped and before the main control program of the FURI core begins executing. The SLUs perform simple ALU calculations that are used when implementing the URISC-style conditional branches that we described in Section 5.1.2.1. The kernel circuits themselves are not directly wired to the datapath of the FURI IEU, we use the standard FastMap mechanism to access their input and output registers. However, their use as primitive, low level operations in system operations such as branching advocates that we consider them as something other than standard application SLUs. We may consider the kernel circuits as "system" SLUs, but the term, as defined in Chapter 4, refers to a different context does not directly translate onto the rôle kernel circuits play: the circuits themselves do not access any privilaged resources within the host array. Rather, we use the term kernel to indicate that the circuits are at the heart of the set of circuits required to support computation within the FURI system.

The FURI system, as implemented, uses three kernel circuits: a 32-bit adder, a 32-bit comparator, and a 32-bit logical-AND. Together, the three circuits provide enough computational facility to implement a branch-if-not-equal operation. In detail, the 32-bit comparator implements the equality test. We use careful circuit floorplanning to contrive a placement of the comparator's output register such that it lies in the second bit position with respect to the circuit inputs. This allows the same map register values to be used for accessing both inputs and outputs of the SLU. Reading the comparator output with the FastMap interface will then return either an integer zero or integer two. The logic-AND SLU is used to mask out any result bits that are not actually part the comparison result. The masked result and proposed branch address can then be fed into the inputs of the

32-bit adder which has the effect of biasing the jump address according to the result of the logical test operation. The biased address presented on the outputs of the adder is then moved directly into the PC, causing the conditional branch to take effect.

Kernel circuits are generally the first circuits to be loaded by any bootstrap or control program executing on the FURI core. This assumes, of course, that we do not include them as part of the FURI core bitstream that is programmed onto the serial PROM. Not taking that approach affords slightly more flexibility overall but, in the early stages of the FURI core debugging, some limited test SLUs where included in the FURI PROM. This was done solely to verify, using an appropriate FURI test harness, that state accesses were operating correctly over the FURI core's internal FastMap interface. Development of the `cal2furi` tool was motivated as a means of systematically loading kernel circuits without relying on the services of any SLUs: to recall, only basic `move` instructions are used in the loader subroutines generated by `cal2furi`.

6.1.5 Assembly Libraries

In the FURI design flow, we see that the FURI assembler receives source files from the SLU point tools and application code created by the system programmer. However, a third set of source files are used by the assembler to actually define and shape the programming environment. Normally, the assembly level programming environment seen by the system programmer is already specified by the hardwired features of the processor architecture itself. For example, the processor may have a defined set of ALU operations and a particular number of device registers. Even subroutine call stack processing will be influenced by the underlying microarchitecture.

This is not the case in the FURI environment. Here, we can define an instruction set interface through the macros contained in one of the standard assembly

library files. On a larger scale, we can define and control, in detail, exactly how the FURI subroutine mechanism operates. The following sections describe some of the main assembly library files that underly the FURI programming environment used by the programs in this thesis.

6.1.5.1 Core Programming Environment Features: Instructions and Constants

A series of assembly library files provide a definition of one core instruction set that is available to FURI programs. This includes the definition of instruction macros which are directly based on the existing facilities of the FURI core. For example, the address of the PC and macros for the unconditional jump instruction '`jmp`' are defined here. Instruction macros and interface constants which harness and represent the facilities of the kernel SLUs are also introduced here. For example, with respect to our earlier discussion of the three basic kernel circuits, we define the three kernel SLU instructions `and`, `add`, and `cmp`. We also define the conditional branch macro alongside the kernel SLU macros because of the close relationship it shares with them.

6.1.5.2 Subroutines

Subroutines are an essential programming construct and their implementation in the FURI programming environment can be slightly complicated. Two forms of subroutine are supported in the initial programming environment: lightweight subroutines; and full-strength subroutines. The primary difference between the two is the amount of context information that is saved and restored between calling and returning from a subroutine.

Lightweight subroutines are defined via two macro instruction definitions, `jsr` and `ret`, and maintain the absolute minimum amount of state information needed to make, and return from, a single subroutine call. Specifically, we define a single location in memory that acts as a minimal call-stack. On making the

subroutine call, we place the address of the instruction that follows immediately after `jsr` in the minimal call-stack. Calling `ret` to return from the subroutine then simply involves moving the contents of the single-cell stack to the PC. At first this approach seems overly restrictive, but it has the advantage of having a very low processing overhead. We only need two move instructions to effect the subroutine call: one to store the address of the return point, and one to unconditionally branch to the subroutine entry point. Returning only requires a single move to unconditionally jump to the return point. Furthermore, the mechanism relies only on the facilities of the FURI core circuitry and there is no need to interact with any kernel SLUs. However, since this approach only supports subroutine calls with a depth of one, it is quite significantly constrained.

Full-strength subroutines allow a deeper nesting of subroutine calls using a call stack implemented in program memory and a frame pointer which traverses up and down the stack as subroutines are called and return. This motion requires the support of the kernel adder SLU to calculate each new frame pointer address. Our use of the adder makes modifying the state of the XC6200's device registers unavoidable. We must enforce the appropriate 'kernel context' on the device before interacting with the kernel SLUs. Specifically, the correct map and mask register values must be set when we use the adder SLU to modify the frame pointer (we can recall from the discussion in Chapter 3 how these registers influence register state accesses within the XC6200). To protect the subroutine caller from these changes, the device context is stored alongside the return address in the call stack. In comparison to the lightweight subroutines, however, this translates to an additional overhead when making and returning from each call.

6.1.6 Challenges and approaches to Loading SLUs

Using the memory-mapped FastMap interface to configure a new SLU onto the cell array is another fundamental operation in the FURI environment. The SLU

loader facilitates this using some of the subroutine facilities introduced above and the output of the `cal2img` point tool. Loading the configuration bitstream of an SLU into the cell array at first appears to be a fairly trivial matter of moving each word of the bitstream into the appropriate section of the FastMap memory map. However, the situation becomes complicated when we plan to use circuitry already resident on the array to effect that configuration. We must address the fact that a bitstream can modify device registers as a valid part of its configuration. For example, we have already seen this characteristic in use to implement the initialisation of the `rpfds` used in the FURI control logic. The discussion of FURI subroutines has already shown how code executing on the FURI core can be influenced by subroutines altering the device state. Here, we must protect the loader subroutine from any modifications to the device state that occur as a side-effect of loading the valid configuration data.

This overall situation raises the interesting question of, what is a valid bitstream? In the literature introducing the two basic models of VC applied some constraints to which array resources a bitstream could access. Only cells within the bounding box of the SLU and, of particular relevance to this discussion, only limited access to the XC6200's control registers would be allowed in a valid bitstream. Two immediate candidate device registers for constrained access would be `config` and `deviceID`. These registers control aspects of the physical format of the XC6200's device interface and should not be altered after the FURI core becomes active. However, the map, mask, and wildcard registers are valid configuration register accesses within a given bitstream since they directly influence the correct instantiation of the SLU circuitry.

The loader subroutine is sensitive to modifications of the device state because it uses kernel SLUs to make dynamic control flow decisions. The specialised subroutines generated by the `cal2furi` point tool are not sensitive to the modification of device registers as they are linear sequences of move instructions that are

executed from start to finish: their control flow is being entirely defined through the sequential incrementation of the PC. However, this approach requires that we also store a specialised loader subroutine for each bitstream, increasing the spatial memory costs. For a generalised loader, the programmer identifies the location of a particular SLU data structure in memory along with the length of that structure when calling the loader subroutine. With that information, the generalised loader subroutine iterates through the SLU datastructure, transferring it to the host FPGA's configuration memory.

To protect the control flow of the loader from changes to the device state, an explicit restore and save of device state occurs before and after any configuration. Specifically, two instruction macros, `save_device_state` and `restore_device_state` allow the programmer to specify a state buffer in memory to which device state is then captured and restored. Two device state buffers are used by the subroutine: a kernel state buffer, identical to that used by the full-strength subroutine implementation; and a CAL-state buffer to retain a copy of the device state created when loading the SLU bitstream. The kernel state buffer is essentially static and is applied after each set of configuration writes. The CAL-state buffer maintains the device state created by loading the bitstream data of the SLU. Since this can change as a consequence of each act of configuration, it is captured after each set of configuration writes and re-applied before any subsequent configurations. For example, if a write changes the value of a map register, subsequent writes in the bitstream may depend on the map register being set to that value. The CAL-state buffer would maintain these map register settings between writes, ensuring that subsequent writes occur within the correct device state.

Having to save and restore the device state amounts to an overhead. However, we can consider amortising that overhead by segmenting the bitstream data into blocks which contain more than one write to the configuration memory, and then applying a block of configuration writes. Still, we should note that we cannot rely

on the services of the kernel SLUs to implement a conditional loop construct for transferring the bitstream. Instead, we dynamically synthesise a sequence of `move` instructions that will, when executed, explicitly move the block of configuration data to the correct set of FastMap addresses. One instruction is generated for each datum in the block of the SLU bitstream being configured. Once the loader subroutine has re-applied the CAL state, an unconditional jump can be made into the code buffer to effect the transfer of configuration data. The last instruction synthesised for this code buffer is an unconditional jump so that, when the block configuration completes, control returns to the point in the loader subroutine which will immediately start to capture the device state.

A small experiment was run to explore the effectiveness of the block based loader with different block sizes. The FURI core was assigned the task of repeatedly loading a 32-bit adder SLU as many times as possible in a fixed time period. The experiment was repeated with successively larger block sizes to see how the effectiveness of the loader changes. Since only the block size is changing between experiment iterations, the count of completed SLU loads at the end of each experiment run is indicative of the effectiveness of the block based loader for that particular block size. We should note here that we are specifically exploring the overheads associated with different block sizes, rather than advocating the arbitrary segmentation of a bitstream into fixed size blocks irrespective of context.

To implement this experiment, a small test harness program was created in FURI assembly language to maintain a counter variable and repeatedly invoke the loader subroutine. Other experiments using a similar harness had already validated the block loader subroutine and we did not concern ourselves with proving that the SLU is indeed loaded properly since this is demonstrable separately. The FURI assembler outputs a memory image of the test harness which is combined with the memory images of the kernel circuitry and then presented to the

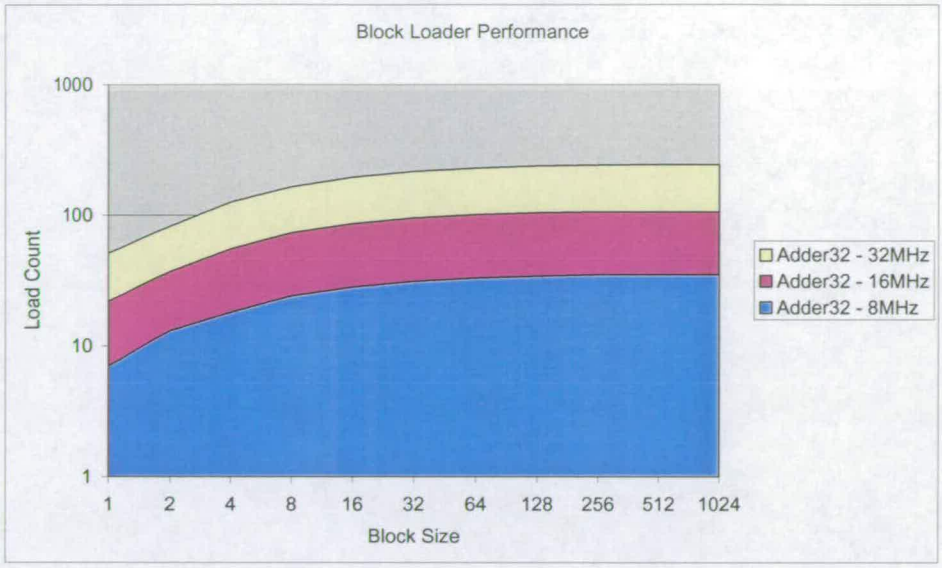

Figure 6.2: Graph of Block Based Cal Loader Performance with Various Block Sizes

`qPCItest` board console program. `qPCItest` is a Linux application that provides very low level access to the features of the Hotworks development system. For the purpose of this experiment, we use a `qPCItest` command script to transfer the application memory image to the development system's onboard SRAM, hand subsequent control of the onboard SRAM to the XC6200, and then initiate a serial download of the FURI core. The FURI core then executes the test harness program, whilst `qPCItest` waits for the fixed time period of one second to expire. At that point the command script forceably triggers a reset of the XC6200 and retakes control of the onboard SRAM. We can then recover the contents of program variables, in particular the load counter, using `qPCItest`'s board memory interrogation commands, augmented with symbol table information generated by the FURI assembler.

Figure 6.2 contains a graph showing the results of this experiment run at three different clock speeds on the FURI core. We show the three experiment variations to demonstrate that there is no disproportionate increase in loader effectiveness

to be gained by simply increasing the physical clock speed of the FURI core itself. `qPCItest` relies on the Linux `sleep` system call to implement the delay between activating the FURI core and resetting the XC6200. However, scheduling variations in such a multitasking environment mean the actual amount of time that a process is suspended by `sleep` can vary. Therefore, the load counts plotted on this graph for each block size are averages of the load count values observed through repeated runs of the test harness for the given block size. The first thing that we can note about this graph is that there is indeed an increase in the effectiveness of the loader subroutine by increasing the block size. The SLU circuit we are loading comprises 615 writes and we can see from the graph that, as we approach block sizes of 256, 512, and 1024, there is little increase in the loader's effectiveness.

We can compare the effectiveness of the block loader against the effectiveness of the `cal2furi` loader subroutine. Figure 6.3 shows the results of a similar experiment to that executed for the block loader. In detail, a FURI test harness program is created to count the number of times that the FURI core can load the same 32-bit adder SLU using the loader subroutine generated by `cal2furi`. The FURI core executes for the same time period used earlier to facilitate a fair comparison with the results from the previous experiment. Again, to reduce any impact from scheduling variations on the delay period, the experiment is run multiple times at each clock speed.

The data plotted in the graph of Figure 6.3 shows the loader count produced on each iteration of the experiment, again at the three different clock speeds. One of the first things we can note about this graph is that the load count magnitudes are significantly higher than those achieved in the previous experiment, even after amortising the state buffering overheads of the block loader. This difference can be attributed to the cost of dynamically synthesising the code buffer instructions used in the block loader. The runtime performance of the `cal2furi` loader repre-

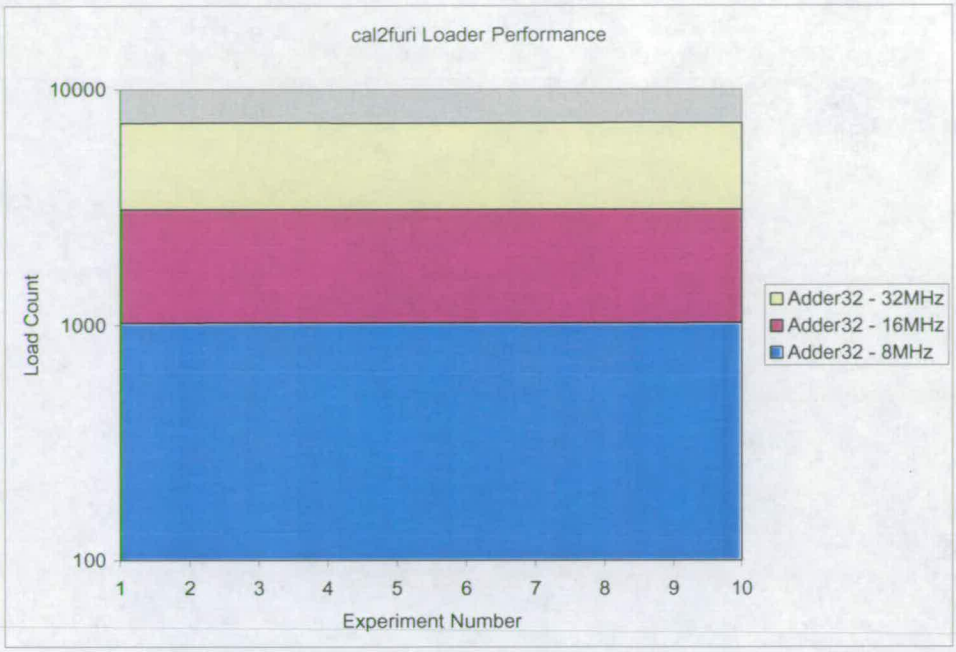

Figure 6.3: Graph of `cal2furi` Loader Subroutine Performance

sents the maximum attainable by a FURI circuit loader implemented solely as a FURI program since it does not suffer from instruction synthesis or device state buffering overheads. The only runtime overhead associated with the `cal2furi` loader is the cost of the subroutine call to invoke it. Whilst the runtime effectiveness of this loader is apparent, we pay for this through the significant static spatial overhead incurred by accommodating both the bitstream image and its corresponding loader program in memory.

In comparing the two approaches, we can see that there is a significant cost associated with synthesising the block loader's code buffer instructions. Whilst this cost cannot be eliminated, it may still be possible to make the block loader more effective by helping it more rapidly amortise the state buffering costs. The block loader discussion assumed a static block size and split the bitstream data accordingly. However, this essentially disregards any potential structure within the SLU bitstream. An interesting alternative approach amounts to an *adaptive* block size. Here, the block size naturally adapts to fit the underlying structure of

the bitstream data in the hope that the bitstream naturally segments into a few large, but irregularly sized blocks which are delimited by harmful device register accesses. If this is true, we can load each large block in its entirety and benefit by eliminating the need to capture and restore device state between block loads since we know that the influence of the device registers does not extend beyond the edge of the block. The only overhead that remains would be the need to reapply a kernel-state since, similarly to the static block approach, we cannot be sure that the device has been left in a suitable state after the adaptive block has been loaded.

To advocate this approach further, however, we must perform an analysis of the structure of some actual SLU bitstreams. What we are particularly interested in is exploring the relationships between parts of the configuration bitstream which modify the device state to those which actually effect changes to cell configurations. For example, investigating what fraction of the overall bitstream is actually involved in modifying the device state and then considering how these writes are actually distributed throughout the whole bitstream. It is worth noting that the open architecture of the XC6200 is key in facilitating such an analysis since we must understand in detail what different parts of the bitstream data are actually responsible for.

The first set of analyses that were run were used to determine the relative distributions of different address types in the bitstream. To recall, the bitstream of the XC6200 is comprised of a series of address and data pairs. The upper bits of the address reveal whether the data will be written into a device control register, or into a region of configuration RAM directly influencing a cell structure. A series of adder SLUs and SLUs from an implementation of the Data Encryption Standard (DES), which is discussed later in Chapter 7, were analysed. Both sets of bitstreams were generated using the standard XC6200 place and route tool. The first graph from this analysis, contained in Figure 6.7, shows the address

distribution for a sequence of adder SLU bitstreams with successively larger bitwidths. The graph shows, for each SLU, the collective percentage of 'cell-data' writes: that is, writes applied to the cells, IOBs, and routing switches. Then, a more detailed breakdown of the writes to each type of control register is given.

From this first graph we can see that a large majority of writes in basic adder SLU bitstreams are configuring cell data: even for the smallest adder, over 94% of the bitstream is cell data. However, aspects of the bitstream generation process may colour our interpretation of the graph. The standard place and route tool for the XC6200 actually supports the customisation of the style of SLU bitstream that is produced. For example, the bitstreams presented in this first graph were specifically generated for a 32-bit FastMap data bus.

Figure 6.8 shows an address distribution for the same SLU bitstreams generated for an 8-bit wide FastMap data bus. We can see in this graph that the percentage of cell data writes is lower than that for the same SLUs in 32-bit mode. This is partly due to a proportional increase in the number of distinct writes that are required to load 32-bit wide data values into the 32-bit wide device registers. However, a closer examination of the raw statistics from the analysis reveals that for the smaller sizes of adder, an 8-bit wide data bus can be more effective for encoding the circuit bitstream. For example, the total size of the bitstream data for `adder02` was 174 writes in 8-bit mode, whilst the same bitstream in 32-bit mode requires 245 writes. As the size of the adder increases, this benefit reduces and the 8-bit mode becomes costly for adders with a bitwidth of 16 or higher.

Overlay bitstreams are another variation in bitstream style that are actually critically important to the FURI environment: all system bus SLUs loaded by FURI are generated as bitstreams in overlay mode. The two previous sets of bitstreams contain configurations of not only the adder circuitry, but also to initialise the XC6200 itself. In effect these SLU bitstreams were generated with the assumption that they would be the first circuits placed on the cell array after

it has been reset. Bitstreams in 'overlay' mode, on the other hand, are generated with the belief that the device is already initialised and that other circuitry may be already configured on the cell array.

A third set of adder SLU bitstreams was generated, this time in 'overlay' mode and Figure 6.9 shows a graph of their address distributions. We can see immediately that SLUs in overlay mode have a much lower percentage of cell data writes than either of the two other bitstream styles. Also we can note that there are no writes to the FastMap ID register since we are assuming the register has already properly configured with at least one 'full' SLU bitstream. In the graph we see that over a quarter of the bitstream is now occupied by writes to the XC6200's mask register.

To determine if this behaviour is a feature of the adder circuitry we re-ran the address distribution analysis on a different set of SLUs. Figures 6.10 and 6.11 show the address distribution graphs for two sets of SLUs from an implementation of the DES in standard and overlay modes respectively. The DES SLUs in each of these graphs, with the exception of the SBOX SLUs, are structurally heterogeneous whereas the adder SLUs were all essentially variations on the same structural theme. From the two graphs we can see that, although the magnitudes are slightly different from those we have seen in the adder SLU distributions, the general form of the distributions is indeed the same. The FURI-compatible overlay SLUs again have a high percentage of writes to the device configuration registers, in particular to the mask register.

We can argue that the increase in the mask register usage is directly related to the nature of overlay SLUs. To clarify, a 32-bit write to the cell configuration store for a standard SLU may actually influence more features of the cell than it needs. For example, only one byte of the 32-bit configuration word may actually contain significant configuration data. Imposing the remaining bytes will also influence the other features of the cell. The explicit assumption that the standard SLU is

not sharing the target cells with any previously configured SLUs means we do not have to worry about overwriting configurations from a different SLUs. Therefore, writing the additional bytes is a safe operation. However, the overlay SLU cannot support the same assumption. It is entirely possible that writing the whole 32-bit configuration word would overwrite previous valid configuration data from a different SLU. The heavy use of the mask register by overlay SLUs in comparison to the standard SLUs, therefore, is to protect cell configurations already applied by previous SLUs.

In the address distributions presented so far, we have assumed that all writes to the XC6200's control registers are potentially harmful but this is not the case. Whether a control register access is benign or harmful depends on the degree of strictness that we wish to impose on the assumptions made by the loader subroutine. At one extreme there are writes to some control registers that are fundamentally benign: the FastMap ID register writes are an example of this. The map register, however, is at the opposite extreme. We know that writes to the map register are potentially much more harmful, since they impact on our use of the kernel circuitry. They threaten not just the correct loading of the SLU bitstream, but the operation of the loader subroutine itself.

The mask register, on the other hand, does not influence FastMap state transactions and therefore does not pose a threat to the control flow calculations of the loader. We can argue that the heavy use of the mask register in overlay SLUs should not indicate the start or end of a loader block. However, the mask register setting does affect any cell-data configurations that follow it and, as such, is part of the device state that must be present for the correct loading of the SLU bitstream. If we approach the adaptive block definition conservatively, we would therefore include writes to the mask register as a feature of the bitstream that delimits a cell-data block. The use of wildcard registers within the bitstream can affect the FastMap state transactions but is potentially less damaging to the

loader's control flow. Changing the wildcard register values whilst the SLU is being loaded means that we may broadcast operands intended for one kernel logic SLU to multiple registers but does not stop the operand reaching its intended destination. The main danger is that the operand will also reach input registers of other SLUs and overwrite meaningful values. Whilst this is acknowledged as a potential danger, we can also observe from the raw bitstreams themselves that the standard place and route tools limit the influence of wildcard registers to cell-data configurations[3]. Therefore, we can argue that the wildcard registers are akin to the mask register in terms of their limited potential harm to the loader subroutine.

The address type distributions presented above establish the *potential* for different cell-data block sizes but they do not demonstrate the effect that the device register accesses have on the actual distribution of block sizes within the bitstream. Therefore, the same collection of SLUs was analysed to reveal their actual cell-data block sizes. This was done twice: first, allowing all device register writes to delimit a cell-data block; and, second, allowing only the map register accesses to delimit the block.

Figures 6.12, 6.13, and 6.14 show the actual distribution of cell-data block sizes in the three sets of adder SLUs. Similarly, Figures 6.15 and 6.16 show the distribution of block sizes for the two DES SLU sets. In both cases, any device register access can delimit a block. From these graphs, we can see that there is a striking difference in the magnitude of block sizes in standard SLUs to those observed in overlay SLUs. For example, the block size distributions for standard adder SLUs contain blocks easily approaching lengths of 100 cell-data writes. Overlay SLUs, on the other hand, appear to struggle to reach block lengths of more than 10 for all bar the adders with the largest bitwidths. We can also see

[3]We should note that this does not consider other XC6200 toolsets. However, we discussed constraints on the validity of an SLU bitstream earlier and applying those constraints in this context would prevent the SLU bitstream's influence reaching beyond the bounding box of the SLU itself.

that a similar situation exists in the two DES SLU graphs: the standard DES SLUs often have potentially long cell-data blocks of nearly 1000 writes whilst the DES overlay SLUs again struggle to reach block sizes of more than 10 consecutive cell-data writes.

Whilst the block size distribution graphs give a feel for the kind of block sizes contained within each of the SLU sets, we cannot deduce how often a particular block size is likely to occur. It remains possible that, even in the overlay SLU sets, the larger block sizes are still the most commonly occurring. Therefore, an additional analysis of the bitstream data was run to determine the frequency of the different constituent block types. For the adder and DES SLUs generated as normal bitstreams, the block frequency did not tend towards either small or large block sizes. Figures 6.17 and 6.18 show the frequency of the different block sizes for the adder and DES overlay SLU sets, but excluding the 8-bit adder SLU set. From these graphs we can see that the overlay SLUs do indeed comprise mainly very small blocks. Both graphs show a bias towards large numbers of very small blocks with only a few of the larger block sizes used in each overlay bitstream.

Figures 6.19, 6.20, 6.21, and 6.22 show the block distributions for the adder and DES SLUs in both normal and overlay modes when only the map register accesses delimit blocks. Figures 6.23 and 6.24 also show the frequency of block sizes within the DES SLU sets. For the adder SLUs, there was a very low frequency count for all block sizes. The DES SLUs show higher frequency distributions because the SBOX SLUs in both overlay and normal modes exploit the register resources of the XC6200 function unit to implement LUTs containing each SBOX value. The register rich nature of these designs means there are more map register accesses during configuration. In these graphs we can see that, when we only consider the most harmful style of device register access as a block delimiter, the block sizes tend to be larger and there is less of a bias towards very large numbers of small blocks in the overlay SLUs.

6.1.6.1 Analysis and Conclusions

The major conclusion that we can draw from the analysis and experimentation in this section is that the SLU bitstreams can indeed contain large regions of device configuration that are benign with respect to other FastMap transactions. In making this conclusion we must nonetheless acknowledge an important caveat: the nature of the SLU bitstream can be radically different depending on the parameters that have been asserted during its generation. This is an important caveat since the performance enhancing VC techniques that we discussed earlier in Section 4.4 actually worsen the block-size distributions. Applying mask register based configuration compression, for example, will actually have the effect of decreasing the average block size.

In relation to our search for sequences of benign FastMap transactions, we have found that the block sizes that are observable within the bitstream can be radically different depending on the block-delimiting criteria that we wish to apply. When we took a conservative approach to defining which features of the bitstream can actually delimit a block, then the block sizes we observed in the overlay bitstreams tend to be small. With a less conservative approach, however, the block sizes within the bitstream not only grew, but also began to reflect the structure of the SLU circuitry itself. This is most apparent in the map-delimited block strategy applied to the DES SBOX overlay SLUs: the bitstreams separated into more small blocks than any of the other DES SLUs because of the register-rich nature of the circuitry.

Overall, our main conclusion is positive since it shows that there is promise in using the adaptive loading strategy for SLUs. However, we can also conclude that the analysis has demonstrated that there are some classes of circuit that have a higher proportion of malignant FastMap transactions. As a technique, adaptive block loading is very relevant over more than one FPGA technology. Since the Virtex architecture also has a modal loading strategy, device context

settings within the architecture also encourage us to pursue an adaptive block-based loading approach for future FURI implementations.

The results from our analysis in this section have a direct influence on the evolution of the FURI framework. We have seen that there are different block delimiting criteria that can be applied to circuits depending on their context and usage. However, the choice of when we apply one delimiting strategy over another is of direct relevance to the FURI framework and the toolset which would support the decision making process. For example, although we have seen overlay SLUs that decompose into many small blocks, there is potential to create larger composite blocks for the benefit of the SLU loading strategy. The process of building such composite blocks from a flat sequence of bitstream data is, in itself, non-trivial. A central reason for this is that we must maintain the integrity of the circuit represented by the bitstream whilst potentially re-arranging its sequence to support the loading strategy. This requires a detailed knowledge of the SLU's circuit structure and, for reasons that we explore in the following section, recovering such information from a bitstream is non-trivial. However, the approach described in [49, 65] may be exploited to enhance the FURI framework and toolset with additional, higher level tools that support the selection and compilation of appropriate loading strategies for different SLUs.

6.1.7 Circuit Debugging

In addition to the low level board console, `qPCItest`, and `qInspector`, one other tool for working with SLU circuitry has been designed.

6.1.7.1 QOverlay

We can recall from the earlier discussion in Chapter 2, that the mainstream design-flows for reconfigurable systems are typically very static in nature. Tool support for dynamic reconfiguration remains, by and large, a product of the dynamic reconfiguration research community. However, there are specific problems

associated with dynamic reconfiguration that are not addressed in a traditional hardware/software design flow. For example, the contemporary design tools of the late 1990s generally did not facilitate designs involving the dynamic overlaying of circuitry. In the FURI system, this is a pertinent problem: we would like to know if loading a SLU bitstream will have an adverse effect on the configuration of any other SLUs already present on the array, or on the FURI core circuitry itself.

The existing XC6200 toolset does attempt to cater for dynamic reconfiguration by generating standard or overlay bitstreams. Even although the toolset will try to minimise the impact of configuration through liberal use of the mask register, we cannot use it to directly determine whether two bitstreams can be safely overlayed. One possible approach to solving this problem without resorting to the development of a new tool from scratch is to attempt to place and route all the desired SLUs as a single design. If it is possible to accommodate the same circuits on the array simultaneously, the place and route process will succeed and we can generate overlay bitstreams for the design components. This would give us the assurance that loading one of the design subcomponents when some of the others are already present will not adversely affect the operation of the others. If the place and route process fails, however, we can conclude that the SLUs spatially cannot be accommodated within the array at the same point in time.

There are two assumptions that underpin this approach and limit its effectiveness: first, we assume that we have access to the source netlist of the SLU; and, second, we are assuming that placement and routing is an entirely deterministic process. Furthermore, the approach only eliminates possible clashes between SLUs within the design. To handle the dynamic instantiation and removal of SLUs from the design, we would have to create new designs with each change to the SLU set and reapply the place and route cycle each time. Regarding the first assumption, the FURI design flow presented above includes the possibility of

pre-defined SLU bitstream libraries to which we would not have access to a source netlist. We can consider these libraries as equivalent to 'hard' cores defined for SLI design[56]: instead of having libraries of fixed circuit layouts, we have libraries of fixed circuit bitstreams. For the second assumption, the use of simulated annealing in placement and routing is a potential source of non-determinism. It is possible that two different iterations of the placement and routing process could result in different bitstreams being produced.

To clarify, the FURI core represents an immutable 'backdrop' configuration that we must consider all subsequent configurations as being relative to. However, if we do try to place and route SLUs along with the netlist for the FURI core, then there is no guarantee that we have preserved the same placement and routing for the FURI reference bitstream. This is especially important where we are motivated to overlay SLU circuitry with parts of the FURI core because of a low utilisation of cell resources. We cannot guarantee that the introduction of additional circuitry will not bias the 'underlying' FURI circuitry whilst at the same time we require the placement and routing tools to take into account the resources consumed by the FURI core. Collectively, this is one reason why we cannot pursue the application of incremental differences facilitated by the `ConfigDiff`[68] tool. `ConfigDiff` generates a set of incremental bitstreams from a sequence of full bitstreams. Each of the full bitstreams can be recreated, in sequence, by applying the incremental bitstreams in order over the initial, base bitstream. However, the identical placement and routing of the FURI core in each of the full bitstream images cannot be guaranteed. As such, we cannot rely on `ConfigDiff` to produce incremental bitstreams that only instantiate the SLU circuitry: the incremental bitstreams produced would also make subtle modifications to the FURI core's circuitry. We should note that the self-modifying nature of the FURI core means that it is technically possible for the FURI core to apply bitstream 'diffs' to its own circuitry. Self-modification, however, is both powerful and delicate: in this

case, its application must be very carefully orchestrated to avoid damaging the core circuitry as it runs.

To deal with this problem, `qoverlay`, is a visual tool that was designed so that the system designer can investigate the interactions between SLU bitstreams as they are loaded on and off a simulation of the XC6200. Again, because of the intimate understanding of the bitstream and device's loading mechanism that is required, the openness of the XC6200 architecture is key in facilitating this tool. The tool assumes no additional information about an SLU other than that given in its raw bitstream data. At its heart is a complete software implementation of the configuration memory of the XC6200 that, in particular, performs an appropriate simulation of wildcarding and masking of the bitstream data being written into it. We simulate the underlying configuration RAM rather than applying the bitstream to a real device because the latter approach would only reveal whether if the circuits could be simultaneously accommodated or not. It would not yield information about why they cannot co-exist or, in more detail, which points of the bitstreams collide.

Operationally, the user specifies a series of bitstream files which are each loaded into the simulated memory interface. This produces a set of fully elaborated configuration RAM images, one for each bitstream. The images produced at this stage are independent of one another, but we know that they fully articulate the SLU's desired device configuration since any wildcarded configurations in the source bitstream are applied and any use of the mask register results in the appropriately masked writes to the memory image. The user also specifies a particular configuration schedule for the loading and unloading of each SLU bitstream. The schedule and memory images are then given to an 'overlay engine' which computes a memory image resulting from the loading of each of the independent SLU configuration images according to the configuration schedule. Rather than just containing the final bitstream data, however, the overlay engine

tags each bit in the overlay memory image with one of four states:

- a 'default' state is defined for bits which have not yet been written to;
- the 'written' state is used for bits which have been written only once since the device was initialised;
- a bit is 'safely overwritten' if it is written more than once with the same bit value and has never been 'unsafely overwritten';
- and, finally, if a bit has ever been written with conflicting values it is marked as being 'unsafely overwritten'.

A coloured visualisation of the state of each cell, switch, IOB, and device pad is generated from the overlay image. However, because we compute the overlay image at the bit level, we can also produce a more detailed breakdown of the state *within* each cell, switch, etc. Rather than showing only the raw configuration data for a cell, for example, a 'translation' view interprets the raw configuration data for the cell and presents the *functional* configuration of its features.

As implemented, the tool is effective but has some notable limitations. The highest level of abstraction that the tool operates at is the level of cell features in the translation view. Whilst it is clear how the features of the cell are affected by the change in configuration, it is not immediately clear how a potentially unsafe configuration will actually affect the higher level SLU circuitry. Furthermore, it is possible that a number of false-positive unsafe writes may be identified by the overlay engine. These are produced as a consequence of overzealous configuration by a previously loaded SLU bitstream.

For example, it is possible that part of an SLU bitstream will include a write that affects all 32 bits of a cell configuration when only a subset of the cell features are required. However, we may overlay that cell with another bitstream at some future point which changes some of these non-essential cell features. If the values

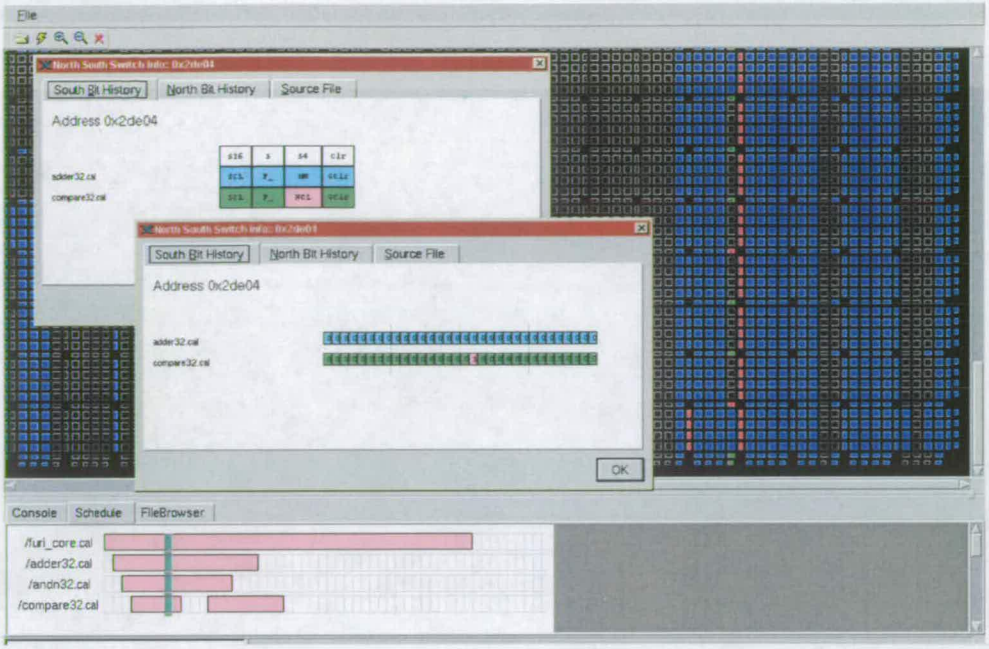

Figure 6.4: `qOverlay` Design Views

written conflict with the non-essential bit-data, the overlay engine would flag the writes potentially unsafe. By convention, the cell features which are written but not required by the the SLU circuitry are given the value zero. In theory, we could use this convention to tag cell features that have the value zero as being 'unused'. This means that if we come to overlay those parts of the cell functionality with another SLU bitstream at some later point we will not conservatively assume that the bitstreams are incompatible. However, from the open nature of the XC6200 architecture, we know also that zero is actually a valid cell configuration and therefore we cannot rely on it as an identifier of non-essential bitwrites.

From our earlier discussion of 'overlay' SLU bitstreams, we can argue that their liberal use of the mask register means that only the appropriate bits in a cell configuration will be altered. In this case, the detailed structural information available within the place and route tool facilitates the generation of such safer bitstreams. However, the FURI bitstream is not generated in overlay mode and we cannot rely on bitstreams generated from all sources to be as precise with the configuration bits they alter. One potential solution to the problem would

be to attempt to recover some information about the circuit structure that is configured by a bitstream. We can use the structural information to determine exactly which parts of each bitstream write are essential.

Even with the open nature of the XC6200 architecture and bitstream, recovering circuit structure from the raw bitstream data is not a simple task. Any value applied to a cell feature is potentially valid, so we must resort to heuristics to increase the degree of certainty that any given configuration bit is actually required by the SLU circuitry. Whilst this approach is not implemented for `qoverlay`, it has been considered in the XC6200 configuration compression work of Hauck et al[65]. Their observation was that they can increase the effectiveness of their compression algorithm if they preserve just the essential features of the original bitstream. The same information could be used in the generation of the final configuration memory image. This allows us to narrow the effect of the write to only the bits that are absolutely essential to the proper instantiation of the SLU and hence reduce the likelihood of falsely identified unsafe writes.

6.2 The FURI Executive

The FURI executive is an implementation of a lightweight co-operative multitasking 'operating system' that executes on the FURI core. The simple test harness programs discussed in the earlier sections use an assembly bootstrap library to initiate their execution. This sort of bootstrapping is very basic, however, and requires the program to manage most of its own execution flow. If we want to perform logically distinct tasks in such an environment, then we need to explicitly code the execution flow for each programmed task. The executive provides a simple multi-tasking framework to allow program tasks generated by the FURI assembler to enter and leave the core's execution flow in a more flexible manner.

6.2.1 Tasks

The existing subroutine mechanism underpins the executive's implementation of tasks. In the basic implementation described here, each FURI task is essentially a subroutine whose entry point has been introduced to the executive. The executive maintains a list of current tasks and, when it has cooperatively received control of the FURI core, schedules a task to run by invoking it as a subroutine call.

6.2.2 Task Switching

In the basic implementation presented here, task switching in the FURI core is entirely co-operative. A task is activated by the executive, but is then responsible for handing control back to the executive when it completes. This has one major advantage: the task is responsible for managing its own context which then reduces the amount of work that the FURI executive must do when giving control to another task. The task list implemented in the cooperative version of the FURI executive is a circular list of task entry points. The executive defines two subroutines that allow the currently executing task to manipulate this list to introduce a new task or remove itself from the task list. The main body of the executive cycles through the tasks contained in this list, implementing round-robin style scheduling.

A pre-emptive implementation of the executive is conceivable but requires additional circuitry support in the datapath of the FURI core. In a preemptive implementation, task switching is triggered on the expiration of a countdown timer. The current implementation of the FURI core does not contain such timers within the IEU datapath, but an enhanced version of the core with an integrated timer is feasible. The integrated countdown timer then facilitates pre-emptive multitasking by periodically buffering the address of the task instruction being pre-empted and forcing the PC to the address of the pre-emptive executive's context switching subroutine. The context switching subroutine must preserve the

state of pre-empted task and restore the state of the task scheduled for execution next. In a traditional processor microarchitecture this context would amount to saving and restoring registers. In the pre-emptive FURI environment, we would have to save not only the XC6200's device state, but also the state of the kernel SLUs.

6.3 Standard System Tasks

Up to this point we have given a comprehensive description of the design and implementation of the FURI core and the fundamental features of its programming and runtime environment. The mechanics of bootstrapping the *circuitry* of the FURI core have been presented, but we have not yet explored the means by which an active FURI core can receive its programming. The earlier description of SLU loader experiments presented one means of populating the FURI core's program memory, but that approach is only really effective within that constrained experimental environment. Also, relying on an external host to provide the initial system programming of the FURI core undermines its autonomy.

This section gives a brief discussion of a fundamental low-level communication model that the FURI executive, as implemented, uses to interact with the external system environment without sacrificing the autonomy of the FURI core. In particular, two FURI executive system tasks that facilitate a more flexible means of interacting with the FURI core are described. Through the discussion of these tasks, we also see how the programming of the FURI executive itself can be bootstrapped.

6.3.1 The FURI base protocol and base protocol handler task

The earlier discussion on debugging the FURI core circuitry characterised the isolating effect of assuming internal control of the XC6200's FastMap interface. In addition to making the circuitry debugging process more challenging, the isolation

of the FastMap interface also affects the way that the FURI core can interact with the external system environment. The FastMap interface cannot be used to stream new programming information to the FURI executive, but the onboard memory of the Hotworks development platform is accessible by both the XC6200 and the PCI card's host processor. This can be used as the physical basis of a simple shared memory model of communication from the external environment to the FURI executive.

The main rôle of the FURI base protocol is to allow the introduction of new FURI tasks into the FURI environment. Specifically, the FURI base protocol provides us with a mechanism to introduce more complex protocol handler tasks and Chapter 7 expands on the nature of FURI communication protocols. A general overview of the base protocol is shown in Figure 6.5. To implement the base protocol, we define a statically sized region of shared memory as a buffer for holding a single base protocol packet. Base protocol packets are written to the buffer by FURI clients and the buffer is periodically examined by a base protocol handler task executing on the FURI executive. One particular aim here is to avoid introducing a significant processing overhead for the base protocol. Ideally, the processing requirements of the protocol are sufficiently lightweight that managing the buffer and processing any packet it contains would not adversely affect the performance of other application protocol handlers resident in the executive.

The FURI base protocol, as implemented, allows a FURI client running on the PCI host processor to write what are essentially 'active packets' into the base protocol buffer. That is, each packet consists of a data section and a code section. Since we are using the base protocol to introduce new tasks, the data section would typically contain fragments of the new task's program code. The code section of the packet contains FURI code defining how the contents of the data section should be processed. Periodically, the base protocol handler examines the base protocol buffer and determines if it contains a valid base protocol packet. If

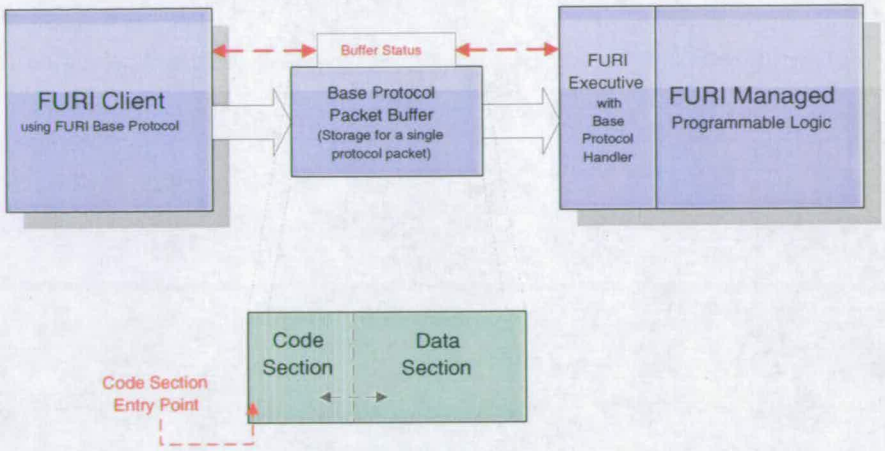

Figure 6.5: The FURI Base Protocol

so, the handler calls the program code contained within the protocol packet as a subroutine.

An explicit status word is used by the protocol handler running on the executive and the FURI client to identify the state of the buffer. Specifically, the status word indicates whether the buffer contains a processed or unprocessed base protocol packet and is used to synchronise the activities of the protocol handler and FURI client. The base protocol handler will only take action if the status word indicates that buffer contains an unprocessed packet. When the code within the packet has been executed, it returns control to the base protocol handler which marks the packet as processed. Similarly, the FURI client only marks the buffer as unprocessed when it has completely written a packet and only fills the buffer when it has been marked as processed. By convention, an empty buffer is marked as processed. It is worth noting that having multiple FURI clients accessing the same base protocol buffer would require a mechanism for granting mutually exclusive access to the buffer.

Using the base protocol to introduce a new FURI task to the executive involves transmission of base protocol packets that are split into two segments: a data segment to contain the new task's memory image; and a code segment to contain the packet's FURI code. The code segment of the packet is aligned at the start

of the protocol buffer to reduce the number of calculations the protocol handler must perform before it can invoke the active code. For the most part, the packet's FURI code transfers the image in the data to a predetermined region of FURI program memory. This could conceivably be programmed as a loop or, since we know the source and destination addresses explicitly, as elaborated sequences of basic move instructions: the active nature of the protocol gives the flexibility to choose. The elaborated move instruction sequence, having no loop overhead, has a better runtime performance but is spatially less efficient and slightly less flexible than a programmed loop. Once the task image has been transferred, the last action of the final packet's program code section, besides returning, is to call the executive's `add_task` subroutine. This inserts the task's entry point into the executive's task list. The task will then be invoked by the executive's scheduler at a later point. Although its primary purpose is to facilitate loading new executive tasks, its active nature makes the base protocol quite flexible. For example, we could use the protocol to load fragments of SLU bitstreams by embedding suitable loader code and bitstream blocks in the active base protocol packet.

6.3.2 The Detacher

Implementing shared access to the Hotworks development system's onboard RAM provokes an interesting programming issue for the FURI core. Physically, the memory used on the Hotworks board is single-ported and uses a wide multiplexor to determine whether the XC6200 or the host PCI processor currently has access to the RAM's address lines. This makes the implementation of a shared memory communication mechanism for the FURI core slightly more complicated since only one of the two potential memory controllers can have access to the actual memory interface at any one point. However, the FURI core has been, so far, presented as reliant on continuous access to the onboard memory for its instruction sequence. Taking access away from the core to allow the PCI host to transfer packets into

the shared region of the board memory leaves the FURI core's memory interface in a dangerous, undefined state.

Whilst this is a specific issue related to this particular development system, the challenge it poses to the current FURI implementation prompts the exploration of embedding executable FURI code somewhere within the host FPGA itself. Allowing the FURI core to execute an instruction sequence contained within the XC6200 can alleviate the core's dependency on the onboard SRAM. The detacher task utilises embedded code to periodically put the FURI core into an 'introspective' state.

When the detacher task is scheduled, it invokes a block of embedded, internal code. Specifically, this code causes the FURI core to detach its interface to the onboard memory for a short period of time. When that period of time expires, the internal code reactivates the board memory interface and returns control to the main body of the detacher task. The host processor can then interact with the shared region of the board memory safely for as long as the FURI core remains detached. The detacher uses a system SLU that wraps around the physical `CON` port providing it with a register accessible interface, to communicate the state of its internal memory interface to the outside world. When the embedded section of the detacher code releases or resumes control the memory interface, it signals a state change over the `CON` port. The low level code in a FURI base protocol client that executes on the host processor therefore monitors the `CON` port to determine when a new packet can be written to the shared buffer.

6.3.2.1 Embedding code within the XC6200's configuration RAM

We can recall from Chapter 2 that the XC6200 architecture, unlike new generation FPGA architectures, does not have embedded block RAM. This leaves two choices for where to embed a FURI program: we can either embed the code as values written to and read from 32-bit wide register SLUs that have been configured on the array; or we can place the program code in the actual, underlying cell

configuration memory of the array[4]. In reality, these are both different regions of the XC6200's memory map that we can access through the internal FastMap interface. Although they have the same access port, they have different access characteristics which justifies identifying them as distinct alternatives.

FURI code embedded within specially configured register SLUs ultimately resides within the cell-state region of the XC6200 memory map. As such it is sensitive to the settings of the device's map registers: if the map register values change as a consequence of the internal code being executed, the FURI core may be unable to fetch the next instruction word or data operand. The linear incrementation of the PC means the register SLUs must be laid out over consecutive rows. The map register settings also constrain the registers to alignment with particular rows. The sensitivity of this approach to the current map register settings, and the fact that the state region of the XC6200 memory map is relatively small in comparison to the cell data region are two significant limiting factors. The map register sensitivity, in particular, limits the embedded code to either only accessing SLUs which are aligned to the same map register settings or orchestrating the placement of the code registers to coincide with changes to the map register values.

We should recall that the onboard memory of the development system operates at a different address granularity to the XC6200's configuration memory. To accommodate this, a slight change is made to the FURI IEU's incrementor. The incrementor dynamically changes the amount that it adds to the PC value based on the memory region that the current PC addresses. In the implementation of the FURI core described in this thesis, the incrementor adds 4 if the address in the PC is a FastMap address, and 1 otherwise. This has an important side effect on the conditional branch mechanism since we must now bias the conditional branch

[4]We should note here that, for XC6200 FPGAs, this is harmless. It is a property of the architecture that loading arbitrary data into the cell configuration memory will not create internal signal contentions. In 2000, this is a property that is unique to the XC6200.

address according to the address space granularity. Effectively, this means that the integer equivalent to the boolean truth value must change depending on the address space.

On the other hand, FURI code embedded within the cell configuration region of the memory map is sensitive to the settings of the FastMap's mask register. This is generally less constraining than the map register dependence of the alternative approach since it does not impose a geographic limitation on the placement of the embedded program. However, we must still ensure that any embedded code does not adversely limit the data that can be read from the cell configuration RAM. Conceivably, we could also reuse the regions of the configuration RAM that control routing switches and IOBs for storing embedded FURI programs.

Any region of the configuration RAM we use to hold FURI code will no longer be available for loading SLUs and the more program code we embed, the larger the geometric region of the array that is consumed. Just as we consider the geometric placement of SLU circuits, we must consider the geometric area of the array that is consumed by storing a FURI program in a linear sequence of configuration words. To do this, again requires a detailed understanding how the underlying configuration RAM maps to the FPGA geometry. A visualisation of the XC6200's memory map is given in Figure 6.6. From the open nature of the XC6200 we know that the total size of each cell configuration is contained in 3 bytes. However, the arrangement of the XC6200's address space means that a single 32-bit word in the cell data region does not completely configure a single cell: rather, it affects a subset of the features in each cell within a particular 4×1 column of cells in the 4×4 cell group that is targeted by the write. The row addressed by the FastMap write to the cell configuration region is contained in the less significant bit positions relative to the bits that define the column address.

Each program word we embed in the cell configuration RAM, therefore, affects a 4×1 column of cells. The first two words of the FURI program, when placed in

two consecutive words of the cell configuration RAM actually affect a column of 8×1 cells. When a FURI program is embedded in the cell configuration RAM, it rapidly affects a column of cells equal to the height of the array itself. This, in turn, may have an adverse affect on the placement of any other SLUs: ideally we wish to localise the region of the cell array that is consumed by the embedded program code. If the entire configuration for a single cell was referred to by a single address within the XC6200's address space then the first two program words embedded in the configuration RAM would only affect two cells. However, it is acknowledged that the configuration RAM was never intended to store linear data such as FURI programs, and that the structure of the address space does actually help to rapidly configure SLU circuitry: the complete configuration for four cells can be packed into three 32-bit writes to the cell region of the configuration RAM.

Rather than consuming an entire column of the array, the embedded FURI code used by the detacher is actually distributed over disjoint blocks of the address space. We preserve the linear execution of the code by threading blocks together with unconditional jumps from the end of one block to the beginning of the next. For example, we can allocate the configuration RAM from a column of 32 cells for holding part of a FURI program. We can then observe that the set of configuration words controlling this column of cells actually comprises three disjoint blocks of the configuration RAM. Most of each block is allocated to holding part of the embedded program code, with the exception of the space for the last instruction in the block which is hard coded to be an unconditional jump to the start of the next block which again contains 'real' program code. We can see that introducing an extra instruction to thread blocks together does constitute an overhead. To balance this we must be careful not to constrain the column size so much that there are so few program instructions per blocks that we spend a large percentage of the embedded code's execution time jumping between the internal code blocks.

6.3.2.2 Alternative application of embedded code

The embedded code of the detacher task demonstrates that it is possible to execute programs held internal to the host FPGA. However, the original, development-system specific motivation for the detacher task does not completely characterise the usefulness of embedded code. A far more compelling reason for embedding code is to facilitate the complete autonomy of the FURI core and FURI executive. To recall, we introduced the FURI executive as the control program that would be executed from the point that the FURI core itself becomes active. The means of supplying the executive's program code, though, was not discussed. However, the discussion above has shown how FURI programs can effectively be encoded as XC6200 *bitstreams*. The bitstream of the FURI core could be combined with a bitstream encoding of the FURI executive and its basic system tasks. Autonomously bootstrapping the FURI circuitry from a serial PROM, as described earlier, would then also load the code for the FURI executive. Altering the PC circuitry so that the hardcoded boot vector for the FURI core points within the configuration RAM would then kickstart execution the executive's program code[5].

6.4 Analysis and Conclusions on the FURI Framework

The major contribution to the thesis of this chapter has been the description of a complete programming and runtime environment for the FURI core. However, we should now consider the effectiveness of the FURI framework for creating reconfigurable computing applications. Throughout the course of the research programme, the FURI framework has been used to create applications that ran on the developing FURI core. As the FURI core's implementation evolved, so to has its associated toolset.

[5]typically, a small embedded bootstrap subroutine would transfer the embedded executive code to the onboard memory to free the region of the cell array it occupies for SLUs loaded through FURI protocols.

In this chapter, the FURI framework has been used to create system applications that demonstrate the FURI core's ability to support the two fundamental operations in a virtual circuitry system: the dynamic instantiation and interaction with SLUs. Furthermore, we also described the operation of the FURI Executive, the FURI base protocol, and detatcher tasks, all three of which have been completely implemented using the FURI framework and constitute the largest system implemented with the toolset. In total, the toolset has been used to create and assemble thousands of lines of FURI code. The next chapter considers the larger research questions that we can address using the FURI Framework, namely at what points we can dynamically adapt the interface protocols executing within the FURI executive to better support the demands of a particular VC application.

Whilst the FURI framework and its toolset have been effective for creating a low level VC environment, applying the framework to larger research questions does motivate the further evolution of the toolset to incorporate the tools and techniques that were described in Section 4.4. The major characteristic of this evolution is the development of a higher level compiler for the description of VC applications. For example, techniques such as configuration interleaving and the TTA specific compilation techniques such as operand sharing are applicable within a FURI compiler architecture. Evolving the FURI framework in this manner increases its attractiveness as a VC application environment. For example, in the discussion of SLU loading strategies, we have seen that there are performance gains to be had by adapting the loading strategy to the features of the circuit being loaded. Applying such an adaptive loading strategy is most sensibly done within a FURI compiler and will make the adaptive loading performance increases available to FURI VC applications.

Overall, the FURI framework will form a high level basis for ongoing research into reconfigurable computing based on virtual circuitry techniques. The next chapter gives an example of one use of the framework as a high-level research,

namely for supporting adaptive interface protocols. An example of further pertinent research beyond that covered in this thesis is the use of the framework to explore meta-configuration languages to support architecture portable SLUs.

6.5 Summary

In this chapter we described a programming and runtime environment for the FURI core introduced in Chapter 5. This began with the presentation and exploration of the components within a design flow and toolset used for programming the FURI core. The programming toolset and design flow was then used to explore different mechanisms for supporting fundamental system constructs such as subroutines. The chapter then presented alternative strategies implemented using the FURI design flow for dynamically loading SLUs. The issues and overheads associated with particular loader strategies were explored and an analysis of the internal structure of bitstreams was given. This analysis provided rationalisations for the inherent complexity and overheads associated SLU loading that had been described in the earlier section. The chapter concluded with a description of the FURI executive, a self-contained multitasking runtime environment for the FURI core. The basic mechanism through which the executive interacts with the external environment was described in the form of a FURI base protocol.

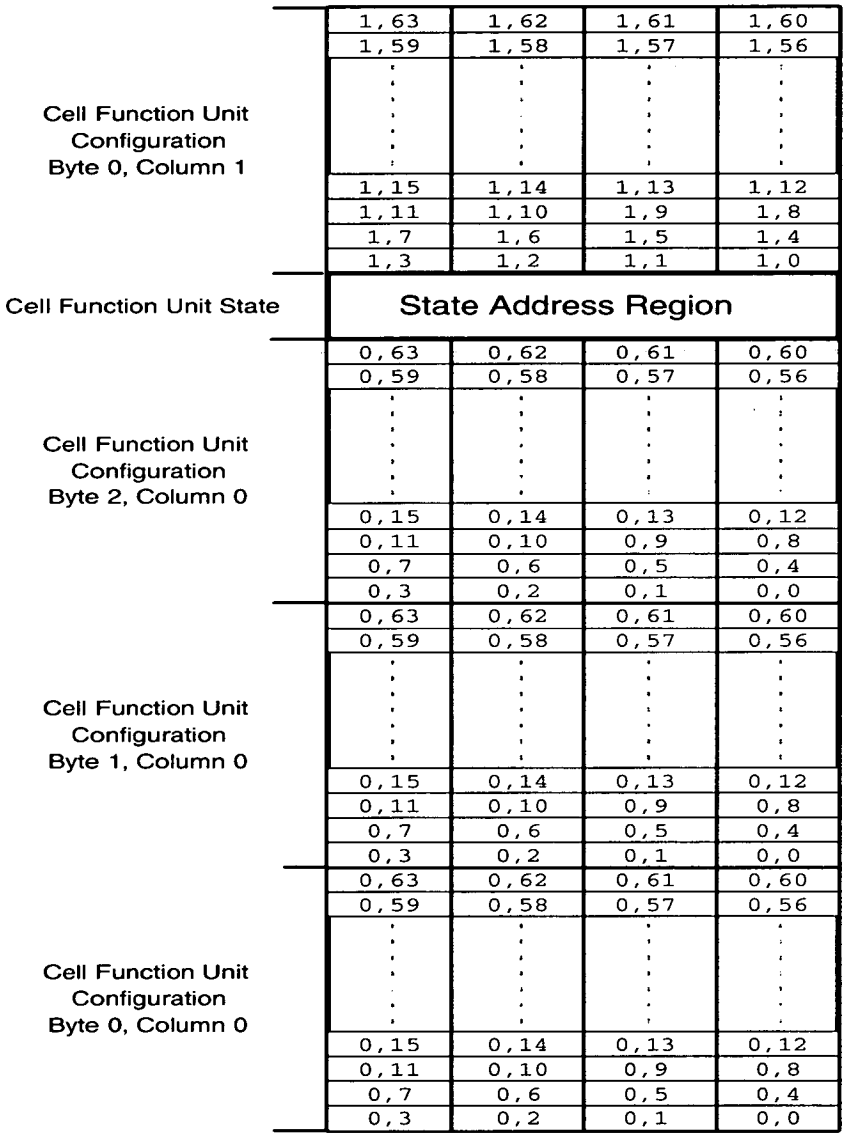

Figure 6.6: XC6216 Memory Map for Cells starting in row 0

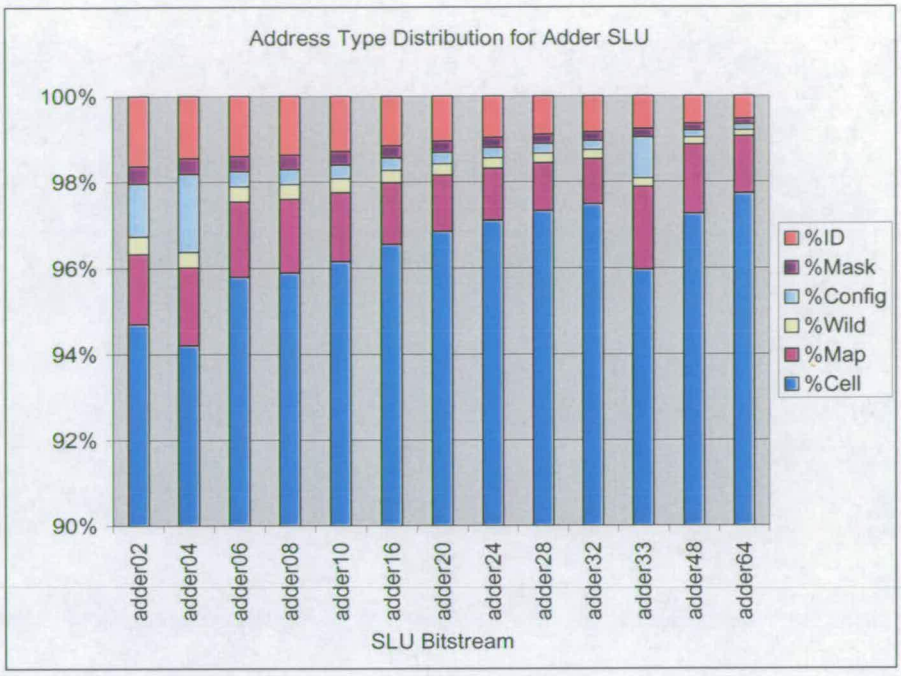

Figure 6.7: Address type distributions in a series of adder SLUs

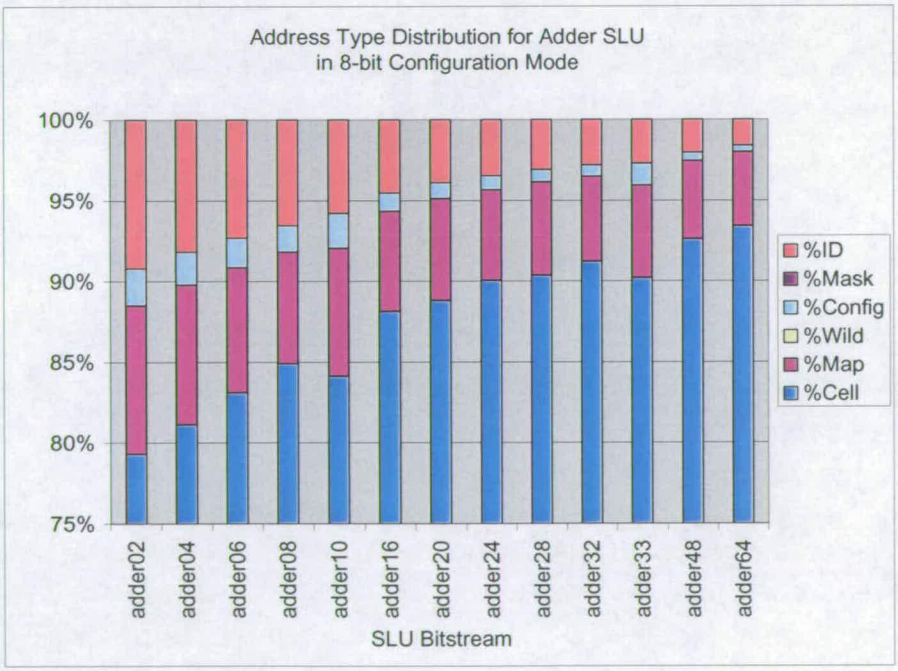

Figure 6.8: Address type distributions in a series of adder SLUs using an 8-bit configuration interface

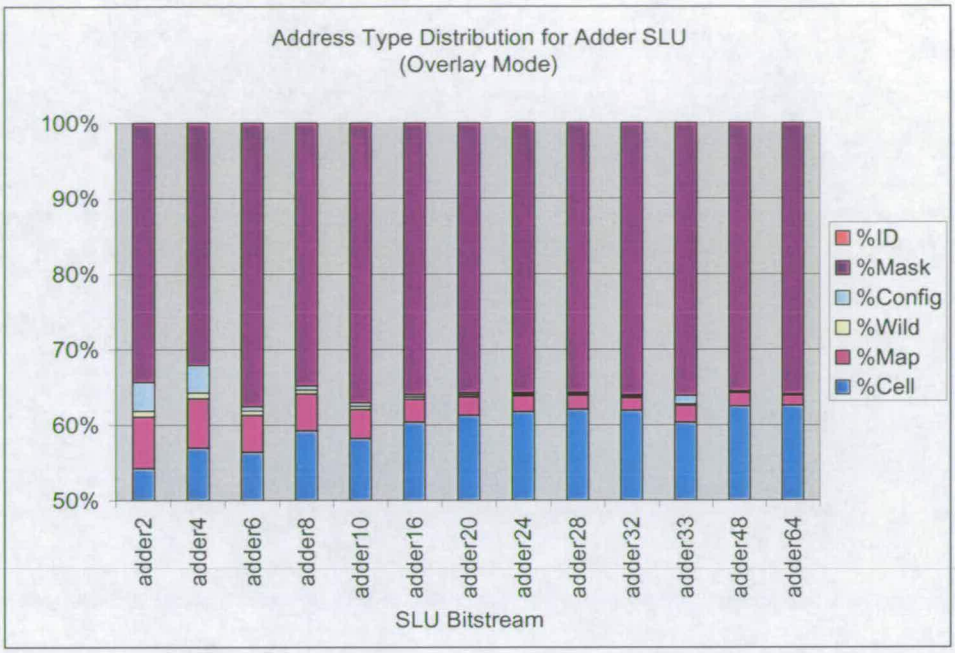

Figure 6.9: Address type distributions in a series of adder SLUs generated as circuit overlays

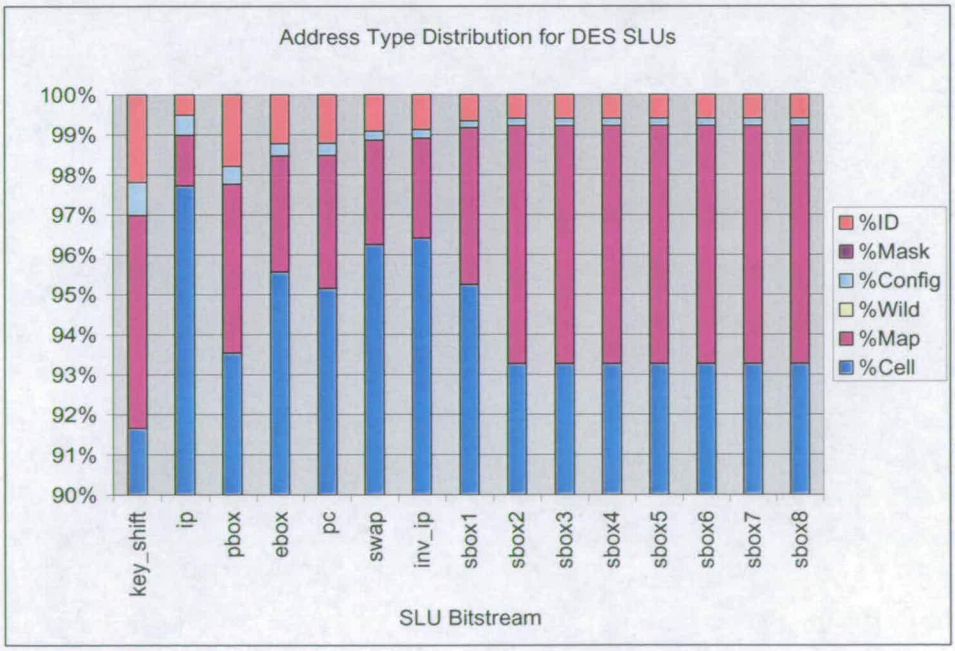

Figure 6.10: Address type distributions in DES SLUs

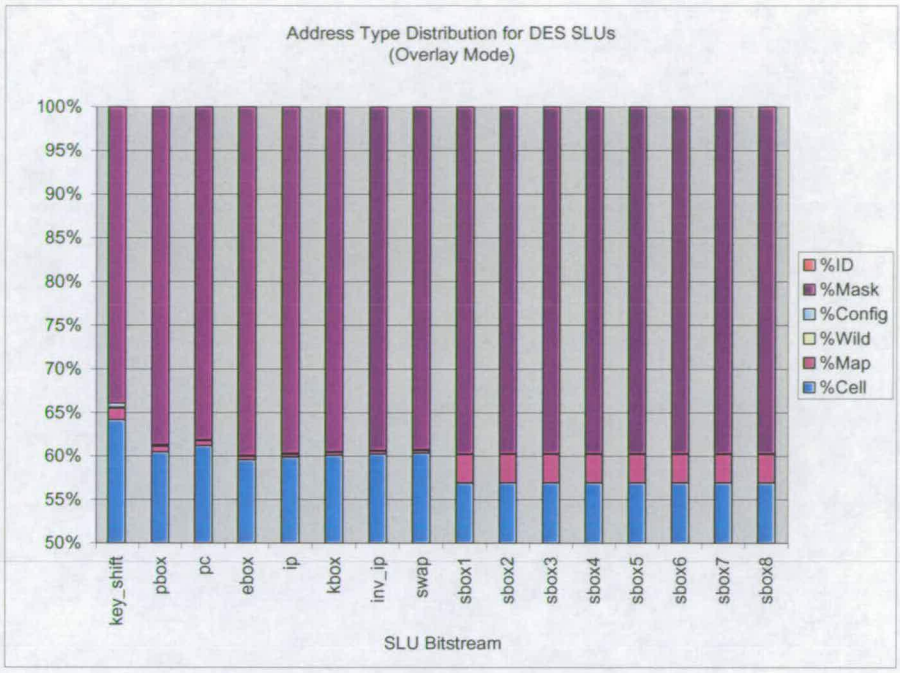

Figure 6.11: Address type distributions in DES SLUs generated as circuit overlays

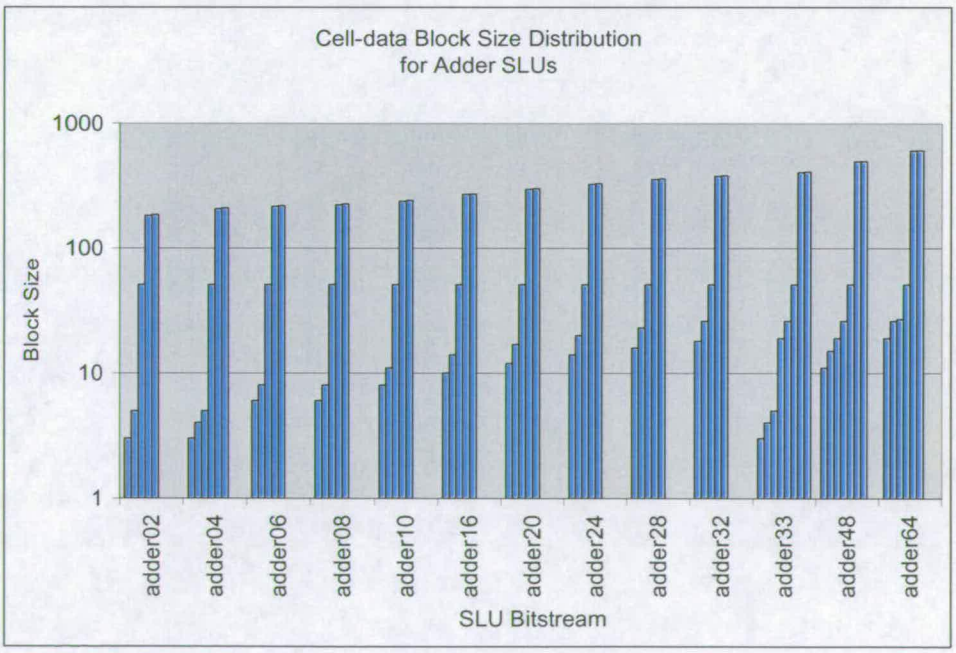

Figure 6.12: Cell-data block size distributions in a series of adder SLUs

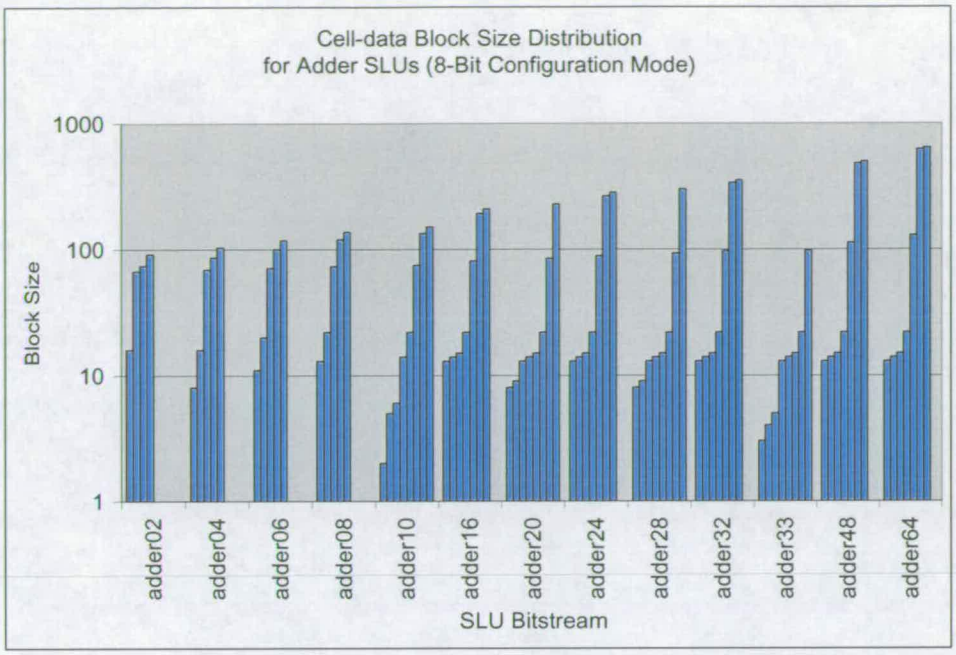

Figure 6.13: Cell-data block size distributions in a series of adder SLUs using an 8-bit configuration interface

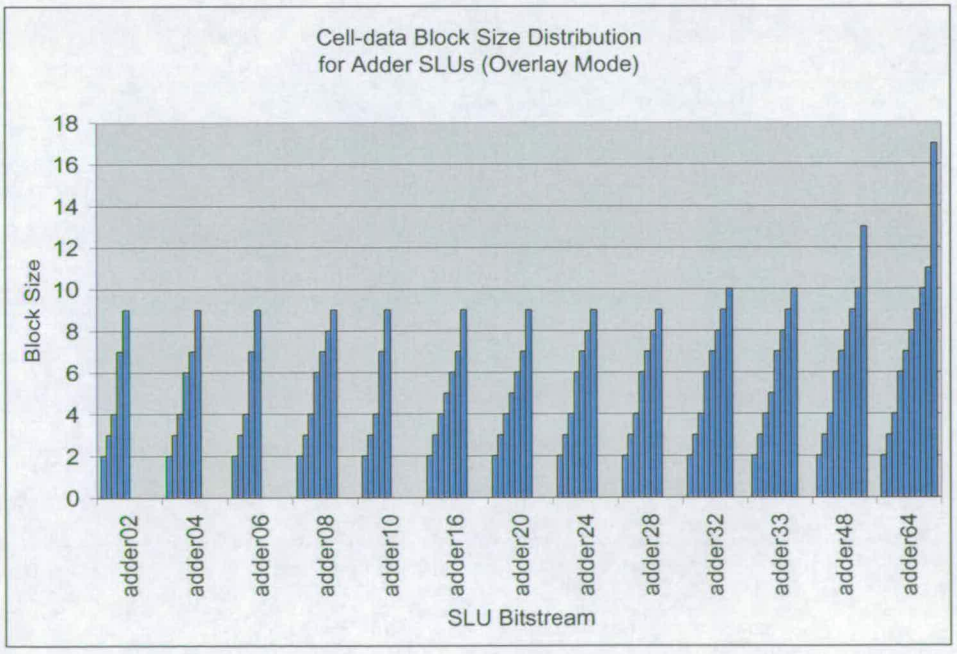

Figure 6.14: Cell-data block size distributions in a series of adder SLUs generated as circuit overlays

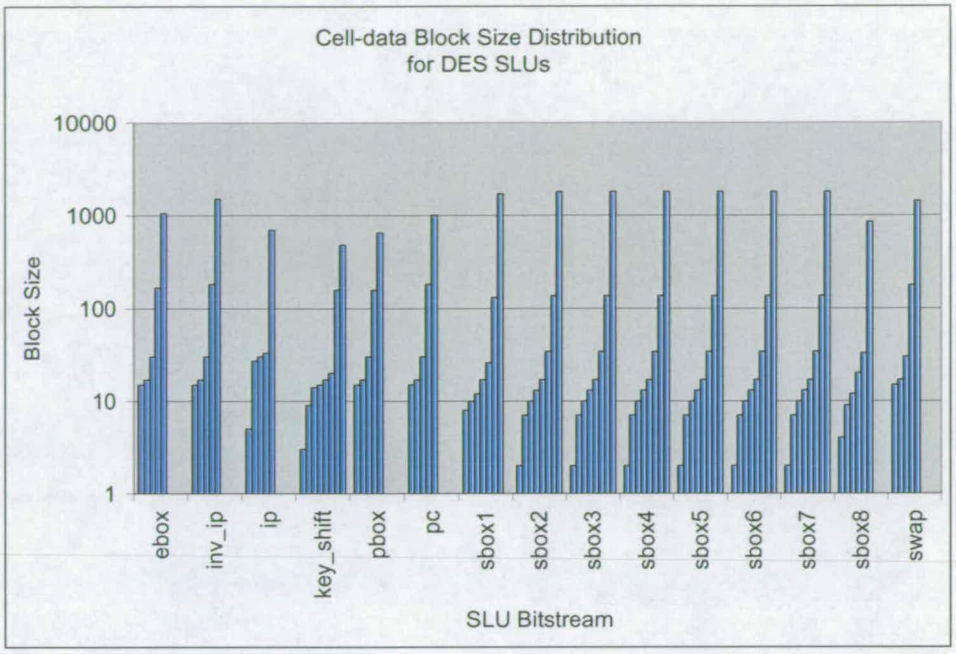

Figure 6.15: Cell-data block size distributions in DES SLUs

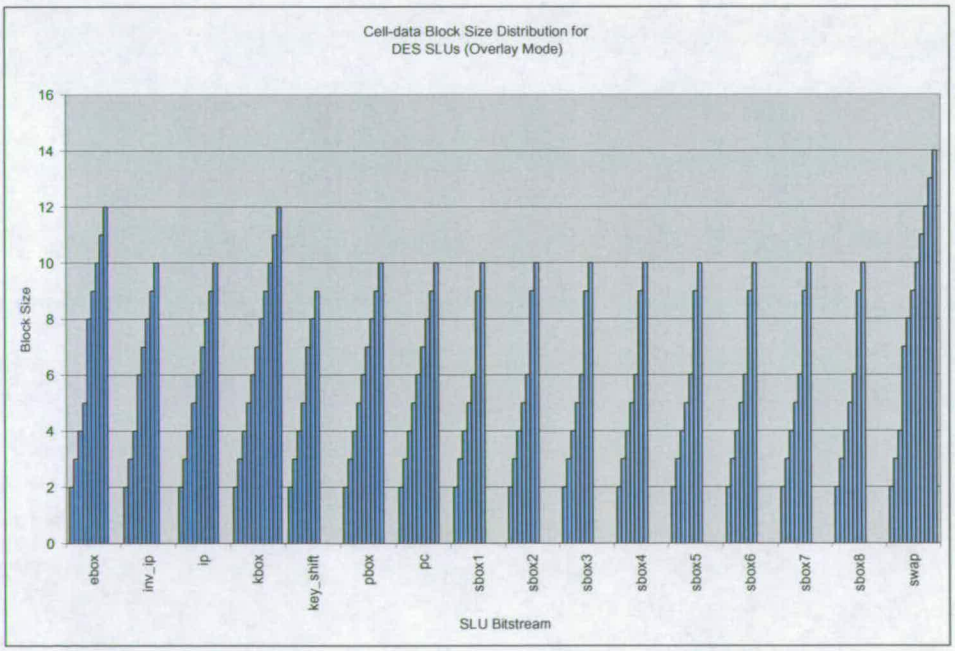

Figure 6.16: Cell-data block size distributions in DES SLUs generated as circuit overlays

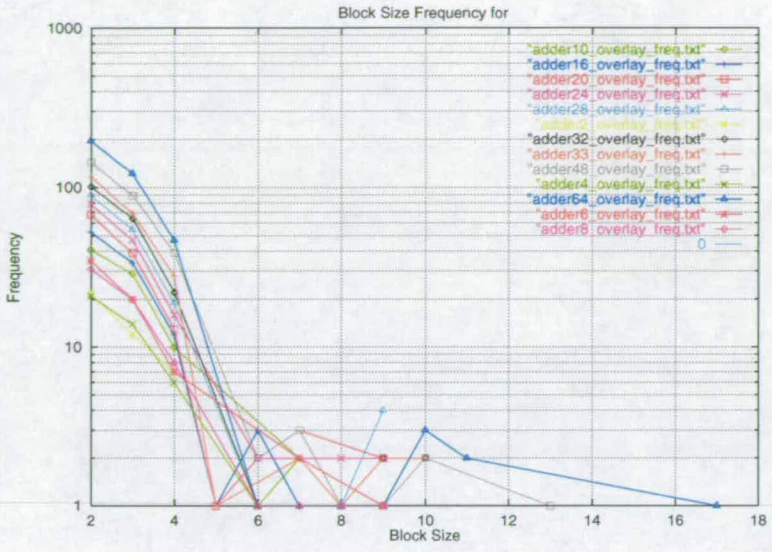

Figure 6.17: Block Frequencies for Adder SLU bitstreams in Overlay mode

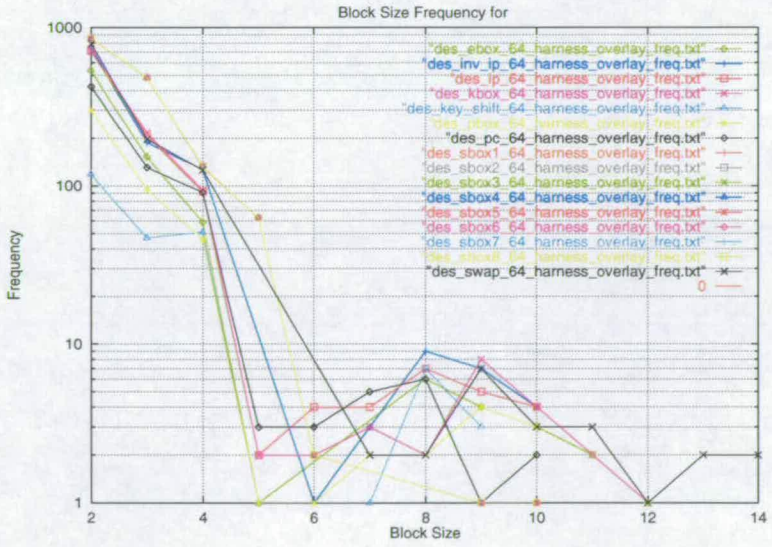

Figure 6.18: Block Frequencies for DES SLU bitstreams in Overlay mode

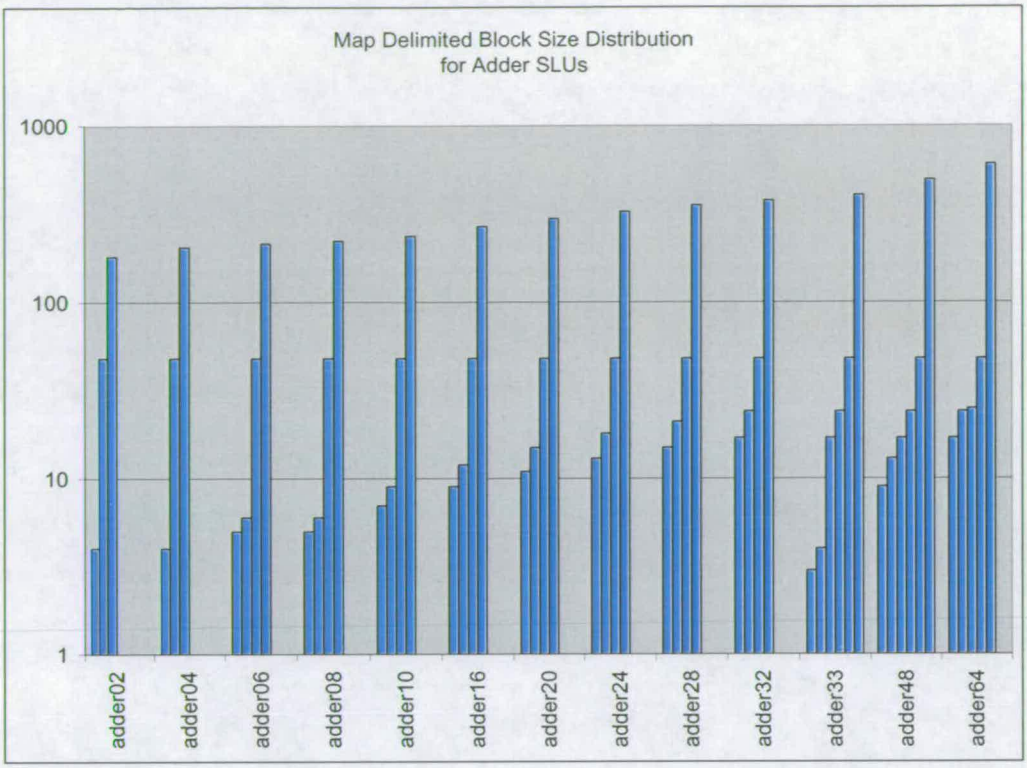

Figure 6.19: Map-delimited Block Sizes for Adder SLU bitstreams

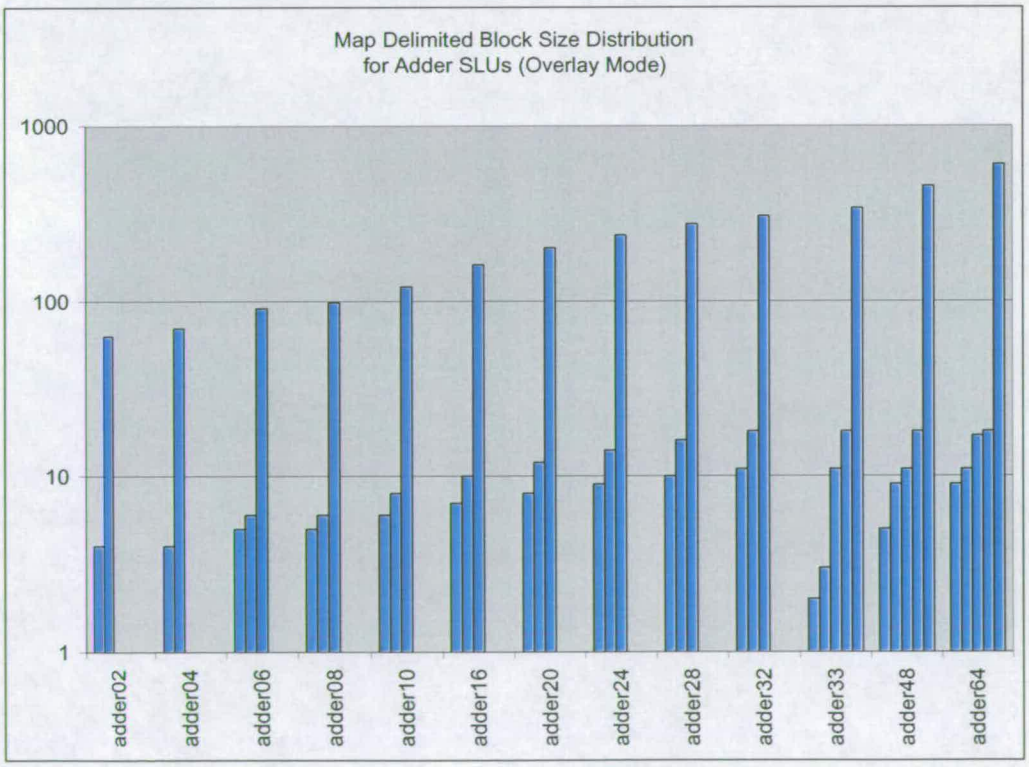

Figure 6.20: Map-delimited Block Sizes for Adder SLU bitstreams in Overlay mode

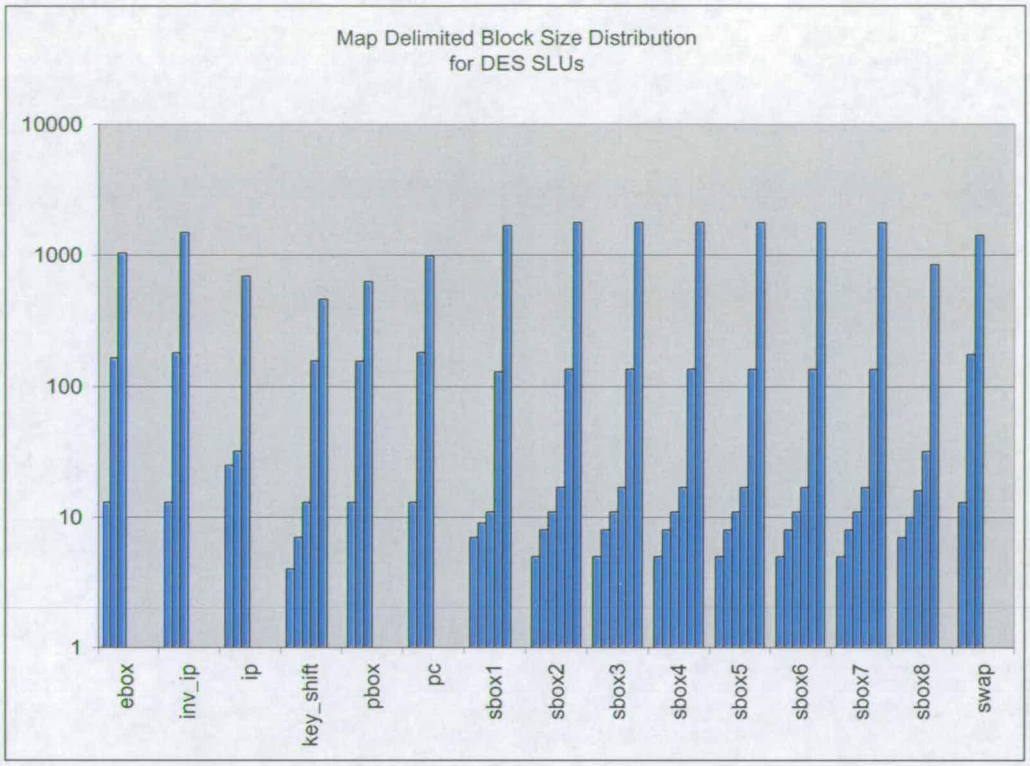

Figure 6.21: Map-delimited Block Sizes for DES SLU bitstreams

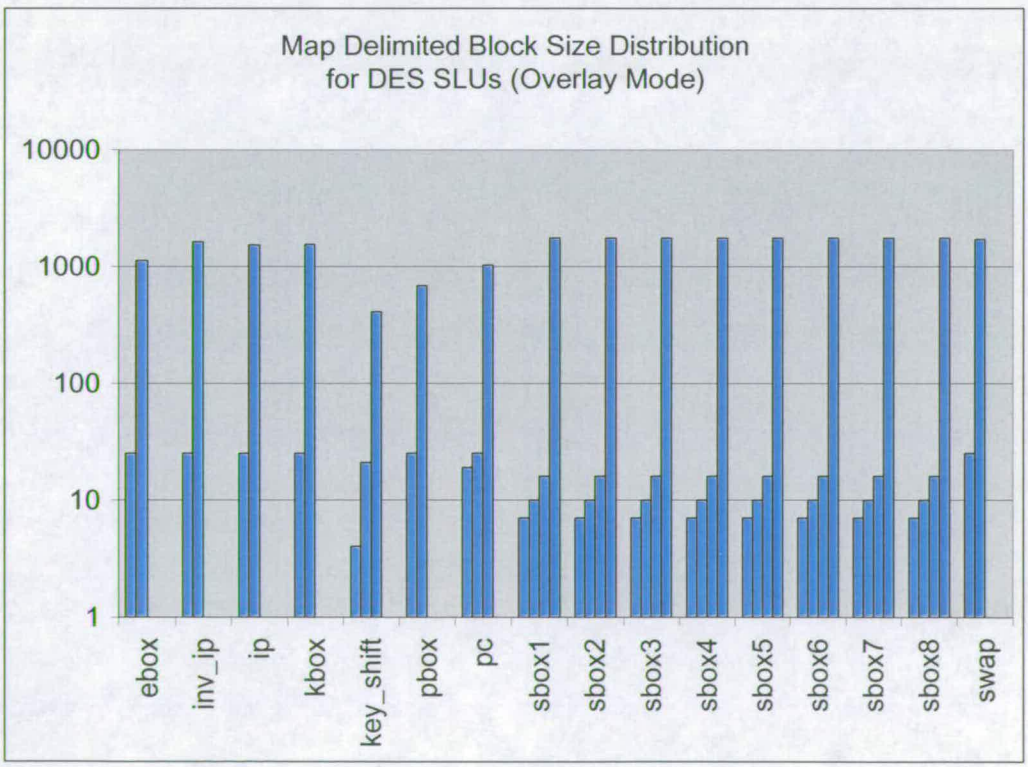

Figure 6.22: Map-delimited Block Sizes for DES SLU bitstreams in Overlay mode

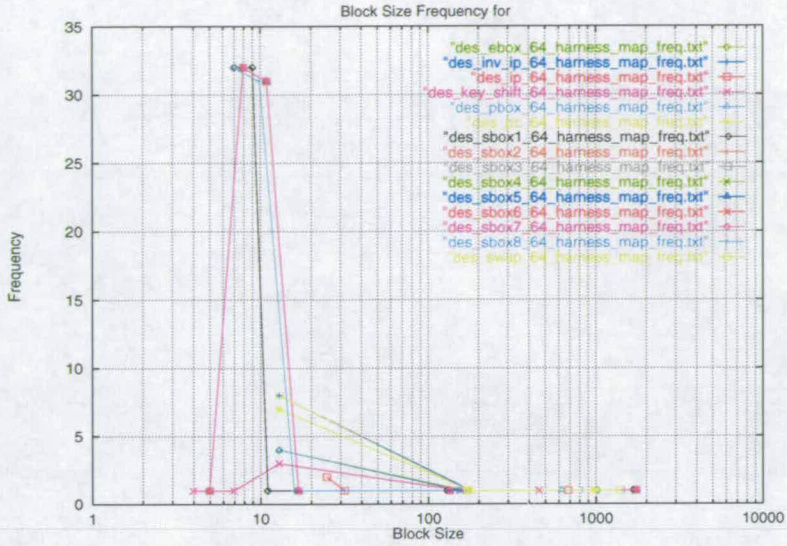

Figure 6.23: Frequency of Map-delimited Block Sizes for DES SLU bitstreams

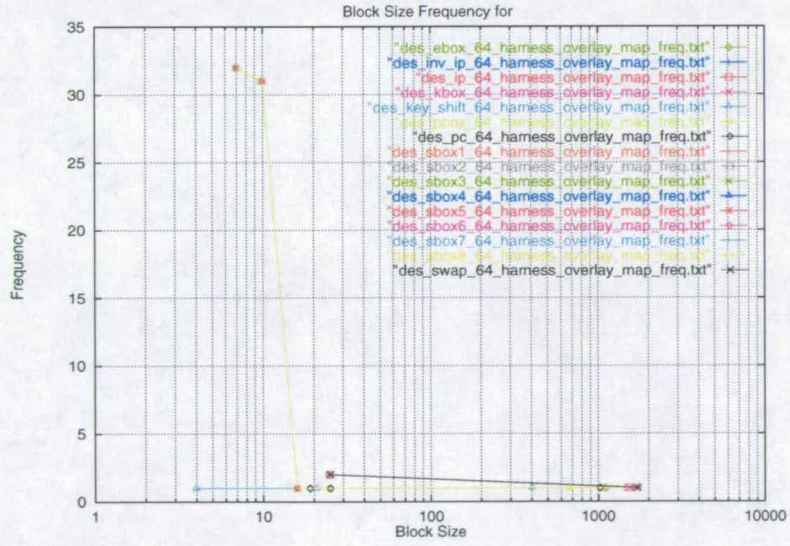

Figure 6.24: Frequency of Map-delimited Block Sizes for DES SLU bitstreams in Overlay mode

Chapter 7

Virtual Circuitry on the Flexible URISC

The aim of this chapter is to explore the implementation of the three different models of virtual circuitry introduced in Chapter 4 on the Flexible Ultimate RISC. The chapter is structured as follows:

- first, we define a FURI system context that the virtual circuitry applications operate within. Rather than simply defining a single context, this section gives a flavour of the diversity of potential FURI system contexts;
- second, we consider FURI protocols expanded from the basic protocol described in the previous chapter, and qualify the design space; and
- third, we present details on, and results from, a related experimental programme. In this programme, we gauge the effectiveness of different protocols used to interface a virtual circuitry application being managed by the FURI core, within a given FURI environment.

7.1 The FURI System Context

Before we can consider the use of the FURI core for implementing virtual circuitry, we must define a particular system context that the applications exist within. In general, we consider the FURI core as operating within a network of co-operating components and each application is partitioned over these components. Figure

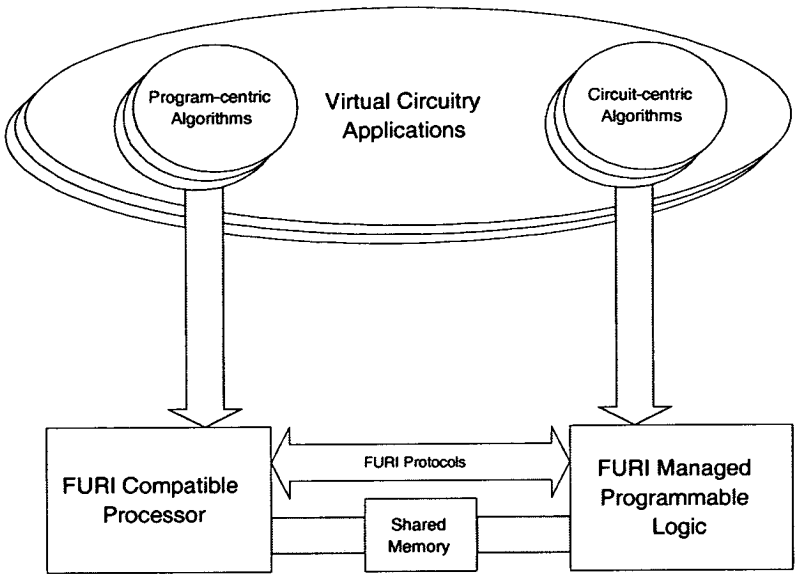

Figure 7.1: Main FURI System Context for Virtual Circuitry Applications

7.1 shows the main system context that we will consider for the experimental work in the later part of this chapter. In it, we see that the algorithms used within a given application are partitioned over a simple network of FURI components. Circuit-centric algorithms are mapped to a FURI network component which is intrinsically circuit-centric, and vice versa for the mapping of sequential, program-centric algorithms to processor elements which are intrinsically program-centric. Application partitioning is generally a difficult problem and the development of methodologies to solve it is beyond the scope of this thesis. We will, however, alude to possible themes for design methodologies for FURI systems in Chapter 8.

The protocols used between the FURI core and other components within the network are influenced by the surrounding network architecture. The simple system context shown in Figure 7.1 does not capture the full spectrum of possible FURI networks and as such will not allow us to present the full variety of FURI protocols. Our motivation for considering different FURI networks in the first place is simple: applications can often operate more effectively, and be partitioned more easily, onto one style network than another. Before exploring the FURI

protocols in detail, it is worth considering the potential different forms of FURI network architecture.

In the following discussions, the term "network architecture" is used to refer to the features of a network holistically: this would include the characteristics of the components used within a network, the protocols used by communicating components, and the network topology. The term "network topology", however, refers only to the geometric and structural attributes of a network. For example, the topological ordering of network components and the geometric classification of the topology are constituent members of a network topology. The logical operation of protocols on a given topology is a separate concern, although there can be clear mappings between topologies and the protocol requirements they prompt.

7.1.1 FURI Network Components

It is possible to distinguish different component types within the FURI network architecture. FURI-managed programmable logic devices and FURI-compatible processor elements are the two primary, active component types. They are the system's computational elements and use FURI protocols to interact through the computationally passive elements of the network. Memories and physical interconnect channels are the two types of passive elements. For each of these four component types, we can identify attributes which influence the network architecture. Some of these attributes are common to all components whilst others may only be relevant to a subset. The following subsections describe six of the primary component attributes.

7.1.1.1 Degree of Connectivity

Here, we consider how many other components a particular component type may be connected to. This attribute can be considered for all component styles.

7.1.1.2 Connectivity Type Constraints

In this case, we are considering constraints on which types of component may be directly connected. In a general sense, this is an attribute of interconnect components as it defines which components may be present at the endpoints of any given interconnect channel.

7.1.1.3 Direction constraints on connectivity

Interconnect channels can be considered to have direction properties. For example, a given channel may support either directed or undirected/bidirectional communication.

7.1.1.4 Communication Mode

The communication mode is tightly related to the above directionality constraints. Some directionalities are mutually exclusive with respect to some communication modes. We can consider a particular channel to support duplex, simplex, or half-duplex communication modes. However, duplex and half-duplex communication require a bidirectional underlying channel. Given a particular mode, it is possible to identify the simplest degree of channel directionality required to facilitate it. Even so, it remains valid to consider directionality as a separate attribute.

7.1.1.5 Synchronisation Policy

Just as we consider the communications mode for a channel, the synchronisation models adopted at the interfaces of network components are also of interest. For example, components could adopt any one of synchronous, asynchronous, or isochronous synchronisation styles. Initially, we will consider synchronisation to be, primarily, an attribute of network component types rather than interconnect channels.

7.1.1.6 Latency

At the low level of abstraction we have been considering so far, interconnect channels themselves have very little physical latency. However, two potential sources of latency exist in the FURI network environment. As data flows through active, computational components like processor elements, a degree of computational latency will most likely be introduced into the datastream. However, the main source of communication latency in the datastream arises from the use of memory components as communication buffers. Specifically, a memory component and the physical channels which connect it to other network components can be considered as a form of compound communication channel. Relatively speaking, the physical interconnect channels have no latency and it is the memory component sandwiched between the interconnect channels which acts as a variable-delay element and may potentially introduce latency.

7.1.1.7 Discussion

In the above sections, the motivation for constraining attributes to a subset of the possible network components may not always be clear-cut. When considering the connectivity type constraint, an underlying assumption is that all communication happens over an explicit interconnect channel. However, there is an alternative to this model: communication may be implicit between components which are physically adjacent. The interfaces of such components may be directly abutting therefore creating an implicit communication channel.

The key point here is the distinction between a form of explicit interconnect and a much more implicit form of connectivity between components. Explicit interconnect channels are those which support communication between components using FURI protocols. Implicit interconnect could occur in a FURI system when we consider, for example, a static slave accelerator attached to either a processor element or a FURI-managed programmable logic device. We do not

consider the slave accelerator to be a processor element in its own right as it does not interface to the rest of the system through a FURI protocol. However, we can include it in the network model if we consider it as an adjacent component to one of the primary, active system components. Here the interconnect channel is implicit between the static accelerator and FURI-compatible component. There will undoubtedly be some form of physical interconnect channel between the two elements, but the semantics of that channel do not constitute a FURI protocol. From the viewpoint of a FURI-compatible device, the computational facilities of the slave accelerator are not differentiated from the native facilities of the active component hosting it.

One further point remains to be clarified: we must consider memory as an exception to the above discussion. From a computational point of view, memories implement identity functions. We can consider them to be a very simplistic forms of auxiliary computation logic. Yet, in the network model we have described so far, memory components need not implement a FURI protocol on the interconnect channels that join them with other elements in the system. In reality, this discussion is prompting us to define the rôle of explicit memories in a FURI network more clearly. We have already stated that memories are passive components and hence do not directly contribute to the ongoing computation in the network. Rather, memories exist to facilitate shared-memory network topologies. It is the set of protocol issues, provoked when using shared memory buffers between active components, that justify the explicit notation of when memories are used in an interconnection path. Since we have limited the rôle of memory, we can also constrain some of its possible interconnection permutations in the network: as a passive element, for example, it makes no sense to allow two memory components to be directly interconnected. In such a situation, neither memory element is capable of actively driving the interface of the other.

On a different theme, it is also interesting to consider the motivation for

attributing synchronisation policies directly to physical interconnect channels. From the physical perspective, an interconnect channel is naturally asynchronous. Network component interfaces situated at the channel endpoints provide the mapping of the three potential synchronisation styles onto the underlying physical channel. We could consider a channel as inheriting a synchronisation style from the attached components, but it is also possible that component interfaces may employ different synchronisation policies over the same physical channel. Essentially, this amounts to logical channels being supported over an underlying physical channel. The discussion of such high level, logical architectures is better addressed as part of the protocol and network topology discussions later in this chapter.

7.1.2 FURI Network Topologies

In the previous section we defined four fundamental components in a FURI network and elaborated some of their primary characteristics. The aim of this section is to show how they may be combined to form different network topologies. However, rather than simply enumerating a few different topologies, we highlight how the design choices at work in a FURI network give rise to different styles of network. Through this, it is possible to build a classification of the resulting topologies which can be used to relate classes of topology to the protocol design issues they provoke. Throughout this section we will use graphical representations to demonstrate instances of particular topologies and classes. Figure 7.2 gives symbolic renderings of the four FURI network component types. We use a boxed 'F' to denote the programmable logic devices which contain FURI cores. A boxed 'C' denotes a processor element which is FURI *compatible* (although we may also consider them to be FURI *clients*). In addition to network classification, we can also consider how some topologies may be implemented on existing reconfigurable hardware platforms.

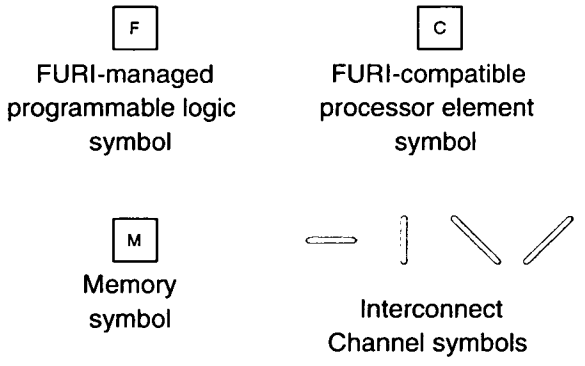

Figure 7.2: Symbolic Representations of the FURI Network Component Types

7.1.2.1 FURI Network Topologies

In network design there are a number of recognised standard topologies. All of these topologies are connected graph structures and examples include simple buses, star networks, ring networks, meshes, trees, and toroidal networks. The type of a given graph is primarily a function of the organisation of its interconnect channels. The organisation of interconnect channels, in general, is closely related to the variation in degrees of connectivity that nodes in the graph may assume. Furthermore, we can identify three main structural classifications within these graph types:

- *regular versus irregular:* ring, toroidal, and mesh networks are examples of regular graph structures. The set of possible connectivity degrees in a graph with regular structure will be small. Furthermore, there is a patterned uniformity in the distribution of connectivity degree within the graph. For irregular graph structures, the set of possible degrees that a node may assume is potentially much larger. But even when the degree set is small, irregular graphs lack the patterned distribution of connectivity degree seen in regular structures.

- *directed versus undirected*: this classification derives directly from the directionality attribute of interconnect channels described earlier. In theory, a

network topology could contain both directed and undirected channels, but more often either one or the other is used. For example, the uniformity of dataflow in parallel algorithms that map well to regular network topologies like meshes also tends to have a homogenising effect on the interconnect directionality.

- *cyclic versus acyclic:* the probability of cyclic structures in a network can be related to an interaction between the set of potential connectivity degrees and particular topological orderings. For example, when topological orderings permit interconnections of node types with high connective degrees, there is a greater probability that two nodes may interconnect to form a cycle. Acyclic topologies, like a star network, have interconnection degree and topological orderings which focus nodes with a connectivity degree of one around single instances of node types with a much higher connective degree. However, even when the topologic orderings and degree spread are uniform and minimal, the directionality of interconnect can still create cycles.

Bridged versus localised is a fourth, conceptual classification and is derived from the constraint on the maximum length of the logical connecting path between any two nodes in the topology. If a network allows two nodes which are not directly connected by a physical interconnect channel to interact, then it is bridged. However, if a node can only ever communicate with nodes to which it is physically adjacent, then it is localised. We treat this as a conceptual classification because it considers the nature of a logical path between two nodes as opposed to the direct structural interconnect architecture.

FURI network topologies conceptually exist in all these forms. Figures 7.3, 7.4, 7.5, and 7.6, show examples of FURI networks in star, bus, mesh and ring topologies, respectively.

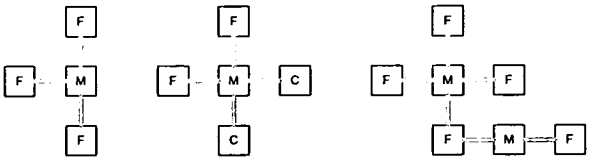

Figure 7.3: FURI Networks containing a Star topology: (i) homogeneous, shared-memory; (ii) heterogeneous, shared-memory; (iii) shared-memory, bridged

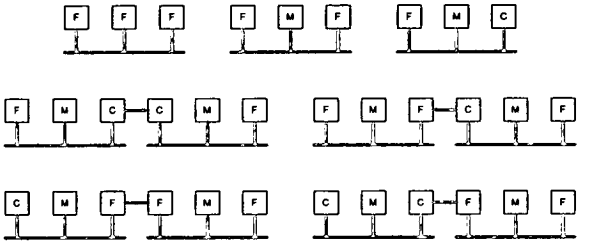

Figure 7.4: FURI Bus networks

7.1.2.2 FURI Network Topological Orderings

By topological ordering, we are concerned with classifying a network topology based on its rules for defining which types of network component can be directly connected with another. The topological ordering of a FURI network is influenced directly by the type constraints applied to interconnect channels. Three FURI network classifications can be derived from the graph topological ordering:

- A *homogeneous* topology, in the context of a FURI network, represents networks where the computationally active elements are comprised entirely of FURI-managed programmable logic devices. Homogeneous networks of FURI-compatible processor elements are conceivable but are not considered. Indeed, our earlier assertion, that a FURI network contain at least one FURI-managed programmable logic device, actually precludes them. The type rules associated with interconnect channels in a homogeneous network only permit the interconnection of FURI network components of the same type;

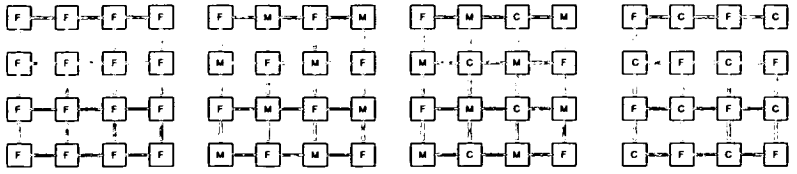

Figure 7.5: FURI Mesh Networks

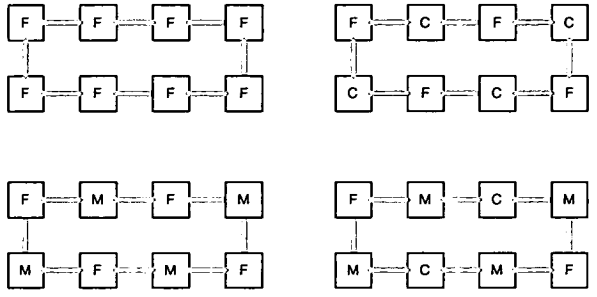

Figure 7.6: FURI Ring Networks

- *Heterogeneous* networks make no constraints on the type of active components in the network. FURI-managed programmable logic and FURI-compatible processor elements may be freely intermixed. The type rules for interconnect in a heterogenous network, therefore, explicitly permit mixed interconnect of different active component types;

- The *shared memory* class is supplimentary to the previous two orderings. In a shared memory network, the interconnect type rules ensure that the computationally active components are only allowed to connect to the computationally passive memory components.

The heterogeneous and homogeneous orderings are, by definition, mutually exclusive. We say that the shared memory ordering is supplementary to them as it is defined in relation to passive memory elements. Both heterogeneous and homogeneous topological orderings can exist in shared memory forms as they only constrain the network's active component types. Neither ordering makes any constraint on the presence nor absence of passive elements in the network. The topology figures introduced earlier also contain permutations of networks in

homogeneous/heterogeneous and shared memory forms. Just as we argued the case for maintaining memories as explicit components in the network model, we can apply derivative arguments to justify the presence of a shared memory class. In the earlier section, we considered the effect of memory components on channel latency. A distinct shared memory class serves as a binding from these issues to the protocol design issues later in this chapter.

7.1.3 Mapping Network Topologies to Existing Platforms

The FURI network models of the preceding sections may at first seem rather theoretical. In this section, we take the discussion out of the purely theoretical domain and consider how these networks can actually be constructed using existing reconfigurable computing platforms. Indeed, the existence of mappings from the logical topologies to existing physical architectures motivates and justifies the broader analysis of protocol issues discussed in the following sections.

The hardware architecture of reconfigurable computing platforms typically imposes a fixed, physical architecture. For example, physical architectures of the SPACE2[80], and Hotworks[1] platforms are shown in Figures 7.7, and 7.8 respectively. Although we may be constrained by an underlying physical architecture in each of these platforms, we may still consider which logical FURI topologies may be mapped onto them. For example, the SPACE2 board physically has a toroidal interconnect architecture which could easily support mesh and ring FURI networks.

The Hotworks development platform has been both the design and experimental platform throughout the course of this research. In terms of a network topology, the physical architecture of the Hotworks development system resembles a simple bus network. From Figure 7.8 we can see the XC6200 and XC4000/Host components share a common bus to the on-board SRAM. Figure 7.8 shows the mapping of a minimal heterogenous, acyclic, shared-memory topology to this plat-

[1] The Hotworks platform is a commercial implementation of the XC6200DS specification[84].

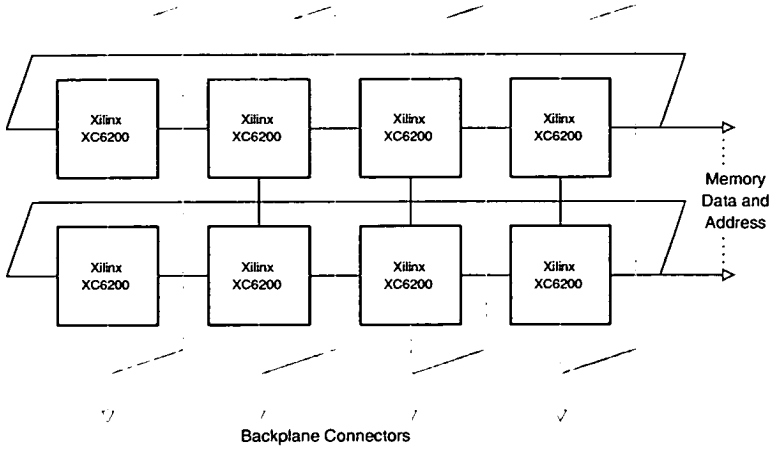

Figure 7.7: Toroidal interconnect of the SPACE2 Computing Surface

form. The logical side of the diagram shows two separate interconnect channels with independent access to the shared memory. This is different from the physical nature of the VCC system where a single-ported memory is temporally shared between the two FPGA components. To prevent signal contention, the memory accesses of either FPGA must be mutually exclusive. The temporal separation of physical bus accesses facilitates the mapping of the two independent logical channels onto the physical interconnect channel. The network is heterogeneous from the point of view that the XC6200 FPGA on the Hotworks board is managed by a FURI core. The FURI-compatible processor element in this situation is the entire host processor system of the Hotworks PCI card. The XC4000 FPGA operates as part of the host processor's interface to the FURI network. It is the host system in its entirety that forms a FURI-compatible processor element. For brevity we do not show the entire host system architecture. The diagram instead shows the mapping of the boxed 'C' symbol in the logical topology onto the XC4000 as a representative of the host system.

The previous sections show that there is a rich design space of FURI topologies. The following discussion of FURI protocols will continue to relate the interesting aspects of the full design space to the issues they provoke. To focus the experimental programme, however, we will demonstrate protocols in the sys-

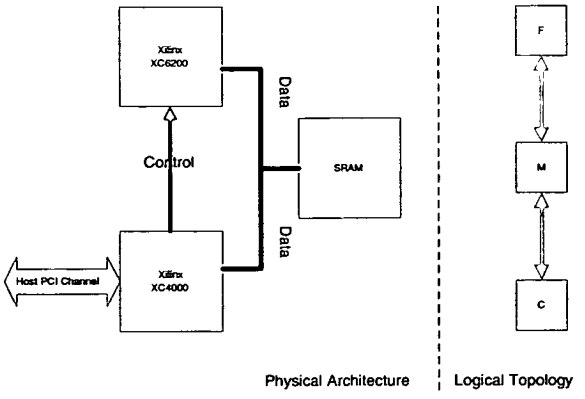

Figure 7.8: Mapping the basic homogeneous, shared-memory topology to the VCC Hotworks Platform

tem context we introduced earlier, with the topology mapped to the Hotworks system. This allows us to first experiment with the fundamental protocol issues between FURI components "in the small". However, insights will still be offered, where relevant, on how the results from the experimentation scale to larger FURI networks.

7.2 FURI Protocols

The rôle of protocol designer is to devise communication conventions which facilitate an efficient and effective transfer of data in a defined network topology. When faced with this task in the FURI system context, the protocol designer, initially, must consider two questions:

- First, what are the communication characteristics of applications which employ a particular virtual circuitry model?
- Second, what are the potential data structures, transmission rules, and transmission mechanisms that can be used to implement the desired communication?

We will use the next sections to consider both of these questions.

7.2.1 Communication Characteristics of Virtual Circuitry Models

In Chapter 4, we defined three models of virtual circuitry: the parallel harness model, the sea of accelerators model, and the sequential algorithmic model. We must understand the communication characteristics of each model before developing protocols to support them. We shall see in the sections that follow how the concept of cohesion within the communication datastream has an influential rôle in our understanding of each model's communication characteristics. The virtual circuitry models we intend to implement can, themselves, be used to implement a variety of different applications. Since the range of applications is not statically defined, we cannot define hard-and-fast characteristics for the three models. We must accept a degree of variability in the characterisations, but need not settle for a complete generalisation.

A large part of our task is to contrast the communication requirements of different models. The more we understand the differences between the communication characteristics of the models, the better equipped we are to advocate particular and alternative protocol implementations. What is equally important, however, is that we also develop of a reasoned understanding of the communication requirements of application classes *within* a particular virtual circuitry model. In the later experimental sections we will compare different protocols for use within a given virtual circuitry application.

The type of data we communicate in a FURI network is common to all three models. The hierarchy shown in Figure 7.9 introduces the four fundamental datatypes in the FURI datastream. Each level in the hierarchy defines an additional interpretation of the raw data in the datastream which is, itself, represented by the root node of the hierarchy.

The first level of the hierarchy shows that we are either communicating an encoding of an algorithm, or *operands* to be processed by that algorithm. It is

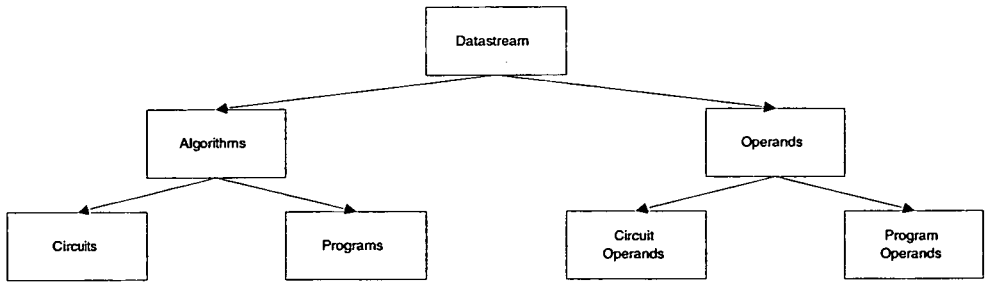

Figure 7.9: A hierarchical rationalisation of types in the FURI datastream

tempting to consider just the transmission of algorithms and 'data' but the term data is too generic to convey the differences in information types we are communicating. "Operand" is much more appropriate term to use: it carries the explicit connotation that something is the subject of an act of processing. At the second level, we use circuit and program as terms which denote a particular encoding of an algorithm. We must also consider the division of operands as being explicitly for processing by circuits or by programs. In the design of FURI protocols, it is important to capture aspects like the distribution of these two operand subtypes. We should also note, though, that there is a raw data format that underpins all four of these data types. Both the FURI core and the configuration memory of the XC6200 operate on 32-bit data words[2].

7.2.1.1 Characteristics of the Parallel Harness Model

In general, the parallel harness operates at a much coarser level of granularity than the other two virtual circuitry models. This applies particularly to the area cost of the circuits and their reconfiguration timescales.

Communication in the parallel harness is dominated by the transmission of circuits and circuit operands and computation is effected completely in the circuit itself. There is no need to augment the circuit with algorithms specified as pro-

[2]We can acknowledge that this is slightly inaccurate since the XC6200 is actually capable of operating over 8-, 16-, and 32-bit data words. However, when the current implementation of the FURI core takes control of the FastMap interface, as described in Chapter 5, it has the side effect of fixing the XC6200's physical interface width in 32 bit mode

grams for the FURI core. Therefore, programs and program operands constitute a much smaller fraction of the datastream traffic in a parallel harness application. The small amount of program operand traffic that does exist in the datastream is for transmitting operands that influence the operating system executing on the FURI core.

Also, there is a large amount of cohesiveness in the parallel harness datastream. The large size of parallel harness circuits[3] means that we must spend a long time configuring them on the logic array before we can use them. Since the parallel harness circuits are also much more likely to be stateful we cannot, as a rule, consider streaming operands through a partially constructed parallel harness. The circuits must be configured in their entirety before any operands are injected. Once the circuit is resident, a long period of operand streaming is required to recoup the cost of circuit configuration.

The cohesiveness we describe above exists within the communication between the FURI core and a single application, or single process within an application. Because of their size, there are likely to be very few parallel harness circuits resident on the array simultaneously: as the circuits are configured, they will very rapidly consume the available array area. Temporal sharing of the array is also unlikely. The costs of a context switch between parallel harness circuits involves reconfiguring the array with the new circuit *and* safely packaging the internal state of the parallel harness circuit being removed. For the majority of FPGA architectures, the cost of implementing such a switch is prohibitive[4].

The conclusion which follows from this is that there are very few applications or application processes trying to simultaneously *configure* the FURI-managed

[3]We argue that parallel harness circuits are large relative to the circuitry used in the other models because they comprise multiple SLUs plus an explicit wiring harness. In this context, the term 'circuit' is referring to the complete collection of parallel harness SLUs and the harness wiring.

[4]Time-multiplexed architectures are an exception here as their multiple configuration planes would support rapid context switches. Most time multiplexed architectures also have facilities for buffering, restoring and sharing circuit context between configurations.

logic. Once a circuit has been configured, however, it is conceivable that the operands being streamed through it could originate from more than one application or application process. The degree of cohesiveness of the operand section of the datastream in this situation depends on the statefulness of the circuit being used. For example, if the parallel harness circuit is a linear pipeline of SLUs, the operands already within the pipeline would not influence the operand being injected at the pipeline input. This kind of parallel harness circuit could be shared between applications and processes on a per-operand basis.

However, if the circuit was a systolic array with internal counterflow paths, the processing of an operand entering the circuit will be affected by the operands that were injected earlier. The results from processing the operands of one application must be protected from the circuit state induced by operands from another application. In short, temporally sharing a parallel harness circuit involves context switching its internal state. The cost of that context switch translates to a performance penalty so enough circuit operands from a single application must be allowed to flow through to make the cost of a context switch statistically insignificant. The cohesiveness of circuit operands in the parallel harness datastream depends on how much state is maintained in the circuit and how far back in the harness that state can influence operand processing. Therefore, parallel harness circuits have the potential for enforcing highly cohesive operand streams which originate from a strictly limited number of applications or processes.

Latency is an important issue in the parallel harness and we will consider it on three fronts: circuit configuration latency; operand processing latency; and result processing latency. Circuit configuration latency is the time between the virtual circuitry application initiating the configuration of a parallel harness circuit, and the point that circuit can be used for operand processing. Configuration latency is generally much larger in the parallel harness than in either of the other two virtual circuitry models. The large size and statefulness of the parallel harness

circuits support this assertion. The main conclusion that we can draw from this is that parallel harness applications expect to experience high configuration latency and will offset its cost by immediately following circuit configuration with long operand streams.

For parallel harness circuits where the state induced by earlier operands affects later operands, it is imperative that circuit operands be presented to the circuit at a specific time. Operands will continue to flow through a parallel harness circuit and the internal state of the circuit will change on each circuit cycle. If we inject an operand too late, the internal state that it depends on to produce a correct result may no longer exist. This provokes the concept of an "operand latency" as the measure of the time between the arrival of operands at the inputs of a given circuit. Similarly, we can consider a "result processing latency" to be the amount of time between each valid result being removed from the circuit outputs. Just as the internal circuit state changes from the constant motion of operands flowing through it, a result may only be present at the circuit output for a particular period of time. If the result processing latency is too high, we will lose results. Our computational efficiency may fall drastically as a result of the large amount of internal state that would need recreated before the missing result could be recalculated.

From this, we argue that parallel harness circuits whose internal state radically influences the circuit output have very strict operand and result latencies that they can operate within. Furthermore, the parallel harness circuits would be very susceptible to variation in the operand and result processing latency, i.e. jitter. In [100], Tennenhouse considers the impact of jitter on protocol design. Operand latency is less of a concern for circuits whose cumulative state does not feedback to influence the result of processing an arriving operand. Only the result processing latency is still imperative here. It remains possible that we will miss the arrival of a valid result at the circuit output if our processing latency is too high. If

this occurs, however, the situation is less severe than for the very stateful circuit. Since the result is not dependent on the recreation of a delicate internal circuit state, it is possible to re-present the circuit operand at the next available circuit cycle.

7.2.1.2 Characteristics of the Sea of Accelerators Model

In the sea of accelerators, many small, independent circuits are used to effect independent computations. This is in contrast to the much larger circuits used in the parallel harness above.

In general, the logic array's geometric area is shared to a much higher degree when implementing the sea of accelerators model than it is in the other two models. Accelerator circuits are small enough for many of them to exist simultaneously within the array. The spatial resources of the array are consumed less rapidly than they are in the parallel harness. Furthermore, the sea of accelerators datastream does not necessarily exhibit much cohesion. Since accelerator circuits are independent units of computation, the act of configuration is asynchronous with respect to the configuration of other circuits: each application or process can begin the configuration of a circuit on the array at disjoint times. The circuit transmissions of one application are therefore intermingled with the circuit *and* operand transmissions of other applications.

Circuit and circuit operands dominate the sea of accelerator datastream. The programs which combine the computationally simple accelerator circuits into much higher level algorithms execute on the FURI compatible processor elements at another part of the network. There is, therefore, little need to transmit programs for execution on the FURI core. The small amount of program operand traffic that does exist is, again, used to influence the FURI executive.

For the sea of accelerators model we must consider the data cohesion at two levels: within the raw datastream itself; and within the logical datatreams of individual applications and processes. Although circuits and circuit operands

are the dominant types communicated, there is very little cohesion in the raw datastream itself. This is a result of having many independent applications and processes sharing access to the FURI-managed logic resource. Circuits from different applications enter and leave the array independently of each other. It is therefore much more likely that, in the raw datastream, transmissions from one application will be intermingled with transmissions from another.

The extent to which transmissions from independent applications are intermingled in the raw datastream depends on the exact protocol in place between the FURI-compatable client and the FURI core. At the finest level of granularity, we can conceive of communication streams being interleaved at the individual operand or single circuit datum level. Alternatively, communication may respect the local cohesion present in a particular application's datastream and only mix datastreams when there is a break in the cohesion of the logical datastream currently being transmitted. A logical question to pose at this point is, "although there is potentially little cohesion in the allocation of the raw datastream to different applications, is there cohesion within the *type* of data being intermingled?". For example, we may only multiplex data of the same type but from different applications, or we may multiplex data of any type and from any application.

In general, there is a high probability that circuits may be intermixed with circuit operands from separate logical datastreams. This is a direct result of the fact there is likely to be other circuitry already configured on the FURI-managed array. Once an application has successfully configured a circuit, it will attempt to stream operands through it. Whilst this is occurring, a different application or process may also attempt to configure its own accelerator circuit onto the array and hence the operand stream from the initial circuit could be interleaved with the circuit configuration stream of the arriving circuit.

Latency is an important issue in the sea of accelerators but for slightly different reasons than it is for the parallel harness. Again, we must consider latency on a

number of different fronts. Firstly, the delay between requesting the configuration of a circuit on the array and the point at which the circuit is configured and ready is *notionally* short. This is primarily because of the relatively small size of an accelerator circuit in comparison to the unit of circuitry used in the other models. We say that the configuration latency is notionally short because multiplexing communication traffic from multiple applications increases the effective latency.

However, in the sea of accelerators model, we do not have to adhere to strict circuit operand and result processing latencies. In their pure form, accelerator circuits are entirely combinatorial and therefore have no persistent state. Since the circuit results are not influenced by any form of internal circuit state, we do not have to worry about presenting subsequent operands within a given time period. The result at a circuit output is a direct function of the circuit's current inputs. We therefore do not have to adhere to a stringent result processing latency provided the current result is removed before new input operands are injected into the circuit.

7.2.1.3 Characteristics of the Sequential Algorithmic Model

The sequential algorithmic model combines operational characteristics of both the parallel harness and sea of accelerators. We know from the discussions in Chapter 4 that, in essence, we are attempting to implement a virtual parallel harness circuit where the FURI core emulates what would have been explicit routing between the processing elements of the harness circuit. Specifically, multiple smaller circuits are transmitted along with a FURI program which defines a *software* routing algorithm.

The first characteristic we note about this model is that programs and program operands play a larger and more central rôle than they have in either of the earlier models. Previous models generally only exploited program operands to conduct a control conversation with the FURI executive. Here, we are not only interested in influencing the existing control system, but actively inserting new

control algorithms on demand. Implementations of both the sea of accelerator and parallel harness models are conceivable were we to communicate an initial control algorithm for the *class* of virtual circuitry model. In the sequential algorithmic model we must communicate a control algorithm not just for the model, but for each circuit presented by an application or application process. These control algorithms, which are rendered into FURI programs and communicated to the FURI core as a program stream, account for the majority of the program traffic in the sequential algorithmic datastream.

The rôle of program operands is potentially quite different in the sequential algorithmic model. Up to this point an implicit assumption has been that the FURI core implements parallelised versions of computations for the sole purpose of accelerating the effective computation on a FURI-compatible processor element. Applications which use a sequential algorithmic model have the potential to reverse this assumption. It is conceivable that the FURI core could transmit program operands to the compatible processor element in the FURI network as a means of using the computational resources available in the network to augment or accelerate the software routing algorithm it is currently executing. When this technique is used, the FURI core implementing the software routing for the current application must wait for the processed result. Sustaining a high level of computational throughput for an application, therefore, requires the efficient communication of these operands and their corresponding results.

Cohesion is difficult to generalise in this model. We inherit the potential for a very cohesive datastream because the computations being implemented are similar to those in a parallel harness: they are relatively large, potentially very stateful, and we are willing to invest significant amounts of the spatial array resource to house them. The knock-on effect from this is that there is little spatial sharing of the resource and, since temporal context switches still incur the recommunication of circuitry, temporal sharing remains too costly to implement.

This implies that we simply transmit the entire computation in one go and follow it with a large enough stream of operands to amortize the setup cost.

However, the sequential algorithmic model is not just an alternative implementation of parallel harness. In the parallel harness we had to configure the entire circuit in one step because the computation itself was specified as one large circuit element. In the sequential algorithmic model, the computation is broken into multiple sub-circuits and program fragments. Provided we maintain the correct transmission order, each computation fragment could be transmitted independently of the other *and* independently of any other sequential algorithmic application running on the array at the time. This is much more characteristic of the situation that we have in the sea of accelerators where the configuration of one circuit can be interspersed with the transmission of operands or circuit configuration for a completely separate application.

In comparison to a parallel harness circuit which effects the same computation, we generally have less circuitry information to transmit for a sequential algorithmic computation. Explicit routes between processing elements are not configured as the FURI core has taken over their rôle. The result of this is that we can actually envisage *more* spatial sharing of the array resource between different applications than we would have considered for the parallel harness. As such, the potential cohesion in the datastream correspondingly decreases. However, and unlike the sea of accelerators, if the amount of circuitry used in the computation increases to consume more of the array resource, then the amount of cohesion in the datastream will increase as a result of the reduced sharing of the array resource.

We can draw the same distinction between logical and actual cohesion in the sequential algorithmic datastream as we made for the sea of accelerators. There is potentially strong cohesion in the logical datastream associated with a single application or process. The assumption that underlies this statement, however, is

that we transmit all the circuitry elements before we can begin to send operands. This is certainly the case for the sea of accelerators and parallel harness, but it does not necessarily hold in the sequential algorithmic model. For example, in the sea of accelerators the entire unit of computation was represented by the accelerator circuit itself and operands could not be streamed through the partially configured, stateless accelerator circuit.

However, in the case of the sequential algorithmic model, the unit of computation is much larger than the unit of circuitry being communicated. Within the logical datastream for a single application circuit, operands may indeed be streamed through the partially constructed computation. As we transmit the first N circuits that are involved in the sequential algorithmic computation, we only need transmit enough of the sequential algorithm's programmed routing to handle operand streaming through those N circuits, yet we still have the opportunity to begin computation after the first circuit and program fragments have arrived. In short, the entire sequential algorithmic computation can be constructed incrementally, with partially computed results streamed and buffered through circuits and the sequential algorithmic control code as they become available.

To summarise, we cannot state conclusively that there is either strong or weak cohesion in the logical datastreams of a sequential algorithmic application. The opportunity to employ interleaved circuit configuration and computation is a compelling reason for deploying the sequential algorithmic model in an application in the first place. If there are a large enough number of circuit operands to follow the complete transmission of all the circuits used by the sequential algorithmic model then we can say that the logical datastream will eventually enter a state of cohesiveness where we are only communicating circuit operands. What we cannot guarantee in this situation is that the circuit operands themselves will all be from the same application.

Latency is also a complicated issue but from our understanding of the rôle of

state in the model, we can argue that it leans more towards the kind of latencies we see in the sea of accelerators than those we argued for the parallel harness. Configuration latency incurred in the model is low because of our ability to interleave the computation of partial results with the arrival of circuitry. The impact of the configuration latency on the initial computation latency is masked since we start the streaming of circuit operands and program code before all of the circuitry for the complete computation has arrived.

Once we have overcome the initial configuration latency, computational latency becomes the time between presenting an operand and receiving the result. The general computational latency of a sequential algorithmic application is longer than the equivalent latency in a parallel harness circuit. This is due to the serialising effect of the FURI core. We generally execute the software routing algorithm step by step and hence, what would have been direct and parallel interconnect in the parallel harness, is now done serially[5]. It therefore takes longer to transform each operand into a result.

The influence of circuit operand and result processing latency varies according to the kind of state used in the sequential algorithmic computation. If stateful circuits are used in the computation, then the application is more demanding in terms of circuit operand and result processing latencies. The same justification that applied in the parallel harness discussion applies here. That is, we must present new operands only when the correct circuit state exists for them to be processed correctly. Similarly, each result may only be present at the circuit outputs for a specified period of time before being overwritten by the results of subsequent operands. We are therefore responsible for ensuring the timely processing of results as they arrive. However, if we only employ stateless circuits in the computation then we know that computational state is managed by the

[5]from the discussions in Chapters 4 and 5 we can argue that this is not a strict rule. Operand Multicasting using the wildcarding mechanisms of the underlying XC6200 means that the FURI core can employ some parallelism in the software routing it implements.

software routing algorithm executing on the FURI core.

Program operands generally require very low communication latencies in this model. This is a direct consequence of their potential rôle in the implementation of the software routing algorithm: if we have long latencies for program operands then the routing algorithm, and hence application performance, must stall until the result arrives. If program operands cannot be guaranteed low communication latencies, perhaps as a result of making multiple hops through the FURI network, we can still conceivably hide their latency by processing a different circuit operand stream. Essentially, we are performing a miniature context switch within the software routing algorithm to a 'thread' which has not reached the stage where it requires the services of the external node. This assumes, however, that there are sufficiently large sections of the routing algorithm that do not interact with external network nodes. Without this assumption, each circuit operand stream would very quickly reach a stage in the software algorithm where it must stall. Therefore in general we consider program operand traffic to have a high transmission priority when it is used in this rôle within the sequential algorithmic model.

7.2.2 The FURI Protocol Design Space

In this section we explore the design space of FURI protocols. There are three main protocol components that we will consider: packets; packet buffers; and protocol handlers. Previously, we considered the communication characteristics of the three main models of virtual circuitry. What we did not consider, however, were the overheads that are associated with each act of communication. Such overhead cannot be completely avoided for all bar the simplest systems, and is influenced by the network topology in which the data is being transmitted. Our previous discussion on communication characteristics allows us to mitigate the effect by tailoring the distribution of the overhead between, and specialising the

implementations of, different protocol components.

The enumeration of protocol components above is actually skewed towards the heterogeneous, shared-memory FURI network that was described earlier as our main system context. That is, the fact that we are considering packet buffers explicitly is a direct consequence of selecting a shared-memory FURI network. This can be interpreted as a high level indication of how the form of a protocol is influenced by the topology of the surrounding network. However, we will not limit the following discussion to the system context alone. Instead, we take each of the three protocol components, discuss their function, and then explore some of their potential forms. We can then relate such forms to the network topologies that require them and the communication characteristics that justify them.

Design choices interact closely between protocol components. Selecting a particular packet style, for example, will undoubtedly affect the implementation of the protocol handler and, depending on the class of network topology, could also impact on the packet buffer format. Enumerating these complex interactions is not our goal, however. We may allude to such relationships but generally avoid detailed discussion.

7.2.2.1 Packet Formats

A FURI packet encapsulates a data payload for transmission in a given FURI network. The packet is the basic quantum of information that is transferred in any one communication *cycle*. We can generalise the structure of a FURI packet into two main sections:

- the packet payload which is the application data being transmitted across the network. The payload in a FURI packet comprises data from at least one of the four fundamental communication datatypes introduced in the previous section; and

- a packet header[6] which contains contextual information regarding the contents of the payload and the way that the packet should be processed by the protocol handler.

We measure the overhead of a packet by the ratio of packet header to packet payload. In theory, the packet header can be eliminated completely. However, this does not mean that we have entirely eliminated the transmission overhead, only redistributed it to the other protocol components. The contextual information excised from the packet header has to be subsequently encoded in either the packet buffer or the protocol handler or, most likely, both. We should note that this is not an all-or-nothing approach. In fact, our ability to balance overhead from different networks and virtual circuitry models comes from the ability to selectively migrate state that would be repeatedly transmitted in packet headers to other protocol components and vice-versa.

A packet can be either fixed size or variable size and the approach we adopt for a given protocol depends more on the communication characteristics of the virtual circuitry model than the network topology we communicate the packet within. The main motivation for adopting variable-sized FURI packets comes from datastreams which exhibit a high degree of cohesion. In this situation we have the opportunity to amortise the overhead of the packet header over a larger section of the datastream: we transmit large amounts of the datastream in a few large variable-size packets rather than segmenting the datastream into many smaller, fixed size packets which each have their own packet headers and corresponding overheads.

Furthermore, the question of whether a packet has a completely fixed *format* is orthogonal to whether the packet has a fixed or variable size. In the most general sense, a packet format is fixed since we do separate a packet into a header

[6]We should note that some network protocols use 'trailers' rather than headers for efficiency reasons. In the abstract sense, both headers and trailers simply correspond to control information embedded within the packet.

which is communicated before the corresponding payload. However, the format of data within those two broad divisions could be either fixed or flexible. Since the format of the packet itself could be encoded in the packet header.

For instance, a packet payload could have contain a mixture of different datatypes or be constrained to only one. A motivating example of this is embedding program data with any of the other three datatypes. This effectively creates *active* packets where the program code contained within the packet actually defines the processing to be applied to the remaining payload. However, mixing datatypes in the packet payload implies that the payload itself has some form of internal structure. If there was no format imposed on a mixed payload packet, the protocol handler would be unable to discern one datatype from another[7]. Therefore, we must consider how to communicate the payload structure to the protocol handler. This can be done by encoding structural information into the header of the packet itself and designing the protocol handler so it can interpret the embedded structural information. With respect to the active FURI packet example, the protocol handler would be unable to discern the entry point of the embedded program unless we encode that information in the FURI packet header.

The simplest form of structural encoding is a type field where we specify the type of a packet in the header and the protocol handler is capable of handling one or more of packet types. This is a fairly static approach as the number of packet types has to be explicitly enumerated before communication begins and the protocol handlers will only ever understand a specific subset of packet types. A more flexible alternative would be to employ a programmed meta-description of the packet format in the header. The protocol handler has to understand where to find the format definition and how to decode that definition but after that it can, conceptually, process any packet format that can be encoded in

[7]It is not possible to arbitrarily mix datatypes in the packet payload since the type of a given datum is not directly encoded into its raw datastream representation.

the meta language. These two examples are at opposite extremes of possible approaches to encoding packet formats. We either have a dense encoding in the form of a packet type coupled with specialised protocol handlers or we define the packet using format specification embedded in the packet header, and have a generalised protocol handler. In terms of their respective impact on overhead, with a meta-language approach we reduce the number of context switches by having few generalised protocol handlers executing on the FURI executive. On the other hand, with a type field there is potentially a larger number of specialised protocol handlers which, as a function of their specialisation, will have a faster processing rate than the generalised handler.

In addition to encoding structural details of a packet in its header, we also convey addressing information. We need addresses in the packet header to identify the node and program or circuit that a packet is destined for within the network. The style of packet addressing required is heavily dependent on the exact network topology and the virtual circuitry model being used. We generally require an address to identify the source and destination of a payload. However, we can reduce the amount of explicit addressing required for each packet by adopting conventions in the protocol handler and packet buffers.

The virtual circuitry model influences packet addressing since the higher the degree of sharing that a model supports, the larger the address space is within a node. Furthermore, the choice between adopting a connection oriented or connectionless scheme is related to the amount of cohesion in the model datastream. A high degree of cohesion generally justifies a connection oriented addressing style since we know that the datastream traffic is going to exhibit a suitable degree of regularity. The most desirable address encodings are compact and have a low processing overhead when the packet eventually arrives at the appropriate protocol handler. It is theoretically possible to eliminate the addressing requirement completely but requires that we severely constrain the network topology and

communication style. i.e., we limit ourselves to a single application or process partitioned over a network where a physically dedicated channel connects two nodes and, more specifically, there is one program or circuit element in each of those those nodes.

Since there are only two nodes in the system context we defined earlier, we can potentially eliminate node addresses from the packets in this style of network. The network has the minimum number of nodes and we need only specify the circuit or program element within the destination node explicitly. At the other extreme, if the network topology and communication style is more complex, the addressing information required increases. In the case of a bridged network, for example, packets are hopping between explicit interconnect channels through intermediate nodes. Essentially we are approaching the complexity of a packet switched network, where intermediate nodes must make routing decisions in the transmission of packets. The processing overheads in such a network make it generally unsuitable for the virtual circuitry applications we are considering but it is interesting to consider as an extreme point on the style of addressing that may be theoretically implemented over the FURI network.

We can also consider the potential difference between packets from a shared memory network and packets from a directly connected network. In a directly connected network, the receiving protocol handler has no real choice in which packet to process since the packets arriving are simply those that have been transmitted from another host. In a shared memory network, there is a greater chance that a packet will experience communication latency by being stalled within a packet buffer. For latency intolerant models like the parallel harness and sea of accelerators the protocol handler must make rapid decisions about which packet to process next. One argument here could be that we simply restrict the class of networks that are suitable for latency intolerant models to those with directly connected nodes. However, forcing all communication to be explicit and direct

between nodes makes sharing of the logic array more difficult and the protocol handlers have to arbitrate direct access to the channel. A shared memory network facilitates sharing of the array by increasing the amount of simultaneous access to the packet buffer. Essentially there is a tradeoff between the overhead cost of arbitrating a direct medium and the overhead cost of processing extra packet data required in a shared memory context. The worst case situation for a shared memory network is a single-ported memory component in place between network nodes. However, this is logically equivalent to the direct connection of nodes since we have to arbitrate access to the memory component rather than channel. For dual ported memories and beyond, the amount of sharing facilitated increases since we have greater independent access to the packet buffers. Here, the protocol handlers executing on FURI components have a larger selection of packets available for processing so the packet may contain extra header information, such as a packet priority or a sequence number, to assist in the decision process.

7.2.2.2 Packet Buffers

Packet buffers exist mainly in the context of a shared memory FURI network and their main purpose is to hold packets that are in transit, between protocol handlers, over a given channel. The FURI protocol handlers insert and remove packets from a particular buffer in order to communicate algorithms and operands between nodes in the FURI network. There are two components that define a packet buffer:

- The first component is a set of operating semantics that define how the FURI protocol handler may insert and remove packets from the region of memory that is allocated for a particular buffer.

- The second component is the context information that must be maintained alongside the buffer to support both the operating convention and the protocol itself.

Generally speaking, there is a correlation between the amount of state we need to maintain for a particular style of buffer and the complexity of its semantics. The more complex buffer schemes require that we maintain more dynamic state to implement the buffer's data structure. Potentially, we could consider a number of different packet buffer styles ranging from first-in-first-out(FIFO) queues, to priority queues, and then fully random-access buffers. The styles are differentiated according to the flexibility with which packets can enter and leave the buffer. For example, FIFO queues allow packets to enter and leave only at given insertion and removal points and only in the order that they were inserted. A priority queue style buffer also constrains how packets enter and leave the buffer but loosens the constraint on the order that packets leave – the lowest priority[8] packet is always the first to be removed, even if it was not the first to enter. In the random access buffer, there are essentially no constraints on how packets would enter or leave. FIFO's are common in communication systems and, in this discussion, we will mainly consider FURI buffers in the style of FIFO queues.

In the context of FURI protocols, FIFO style buffers have two useful properties: first, they implicitly preserve packet ordering and, second, the process of inserting and removing a given packet has $O(1)$ time complexity. From the discussion on communication characteristics, we know that packets, especially from operand streams, arrive and leave frequently. This makes the constant-time insertion and removal of a packet an important consideration. Figure 7.10 presents three variations of FURI FIFO-style buffers.

The different FIFO implementations show that we can, even within the restrictions of the given style, still make some tradeoffs in the amount of dynamic state we support. In particular, the cases in Figure 7.10 show the effects of altering the buffer's granularity of access on its state requirements. For FURI FIFO buffers, the access granularity itself breaks down into two components: the insertion gran-

[8]This is an assumed convention. Different queue implementations may assert that the highest priority packet should be the first to be removed from the buffer.

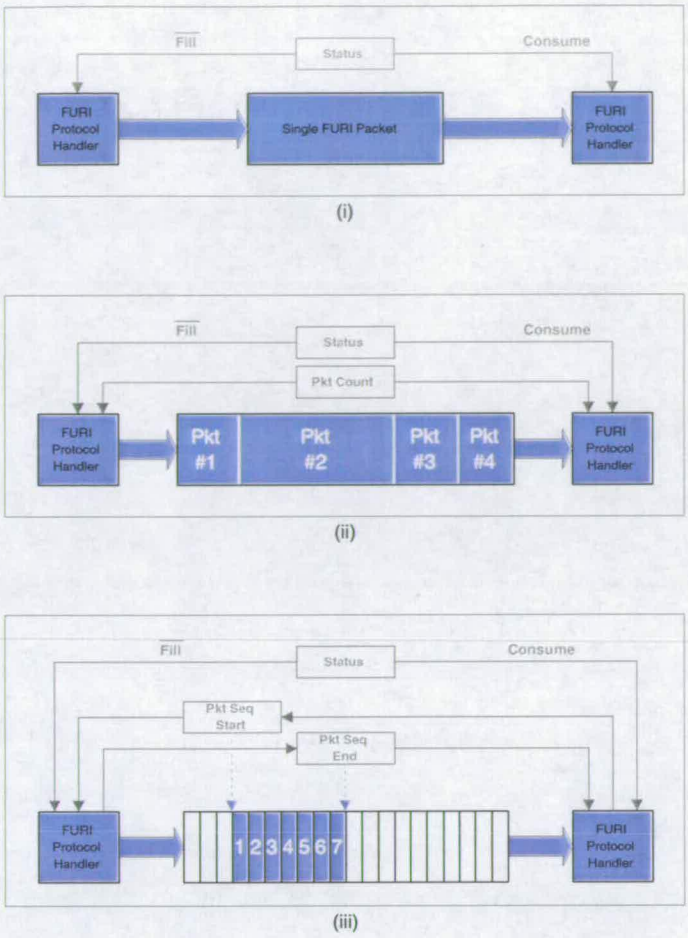

Figure 7.10: FURI Buffers with FIFO style operating conventions: (i) a minimal FIFO buffer containing one packet; (ii) a multiple packet FIFO filled with an access granularity matching the buffer size; and (iii) a multiple packet FIFO supporting single-packet access granularity.

ularity determines how many packets enter the buffer with each access, and the consumption granularity defines how many packets should leave the buffer with each access. It is worth clarifying, however, that we are assuming the memory region allocated to any FURI buffer is itself static and that each protocol handler associated with a given buffer has explicit knowledge of both the memory size of this region and its start address.

Figure 7.10(i) shows the simplest form of FIFO buffer that we will consider, and is essentially the same buffer form that was used in the FURI base-protocol in Chapter 6. In terms of access granularity, this buffer must be either entirely

filled or entirely emptied each time it is accessed. The only state information that we need to associate with an instance of this buffer style is a flag indicating whether the buffer is currently full, or whether it is currently empty. A single packet buffer may, at first, seem quite limited, especially in the context of small, fixed-size packets. However, this buffer style effectively demonstrates the smallest quantum of buffer state required to exchange a single packet between two protocol handlers and would be effective in situations where the VC application datastream is cohesive enough to warrant large variable-sized packets.

Figure 7.10(ii) shows a multi-packet variant of the first buffer. Although the buffer may contain multiple distinct packets, it is still filled or emptied in its entirety with each access. Now, in addition to maintaining the full-empty indicator, we must maintain a count of the number of packets that are currently held in the buffer. Given the current set of constraints, this is actually redundant when the packet size is also fixed. If that were the case, the number of packets could be deduced from the memory size of the buffer and the size of the packet and the memory region allocated to the buffer would be a multiple of the packet size. The packet count is primarily needed to let the receiving protocol handler know how many variable sized packets are held within the buffer. With this information, the receiving protocol hander can then consume the appropriate number of packets, starting with the first packet whose header is aligned with the start of the buffer's memory region.

Finally, Figure 7.10(iii) shows the FURI model of a multi-packet FIFO with an access granularity on the scale of a single packet for both insertion and consumption. For this buffer style, in addition to the status word identifying whether or not the buffer contains any valid packets, we maintain two dynamic pointers into the buffer's memory region. These pointers define the start and end of a valid-packet region respectively. The FIFO operates cyclically, so the pointers wrap around whenever they reach the end of the buffer's memory region.

As with the previous two FIFO styles, the receiving protocol handler starts to consume packets based on the value of the status flag, but now only consumes packets within the valid-packet region delimited by the start and end pointers. When all the packets within that region are consumed, the two pointers align at the same address and the status word must be inverted to show that all the packet data is consumed. This is important because the protocol handler inserting new packets must stall if it wraps around to the start of valid packet data that has not been consumed by the receiving handler. The inserting protocol handler increments the valid region's end pointer each time it adds a packet. Before adding a packet, however, it tests whether or not the valid region's end pointer matches the start pointer and whether the buffer's status indicator is set to show the buffer still has valid packet data. If both conditions are true, then the buffer is full and the inserting protocol handler must wait until either the buffer is marked as empty again or the valid-region pointers no longer align.

In comparison to the two earlier FIFO styles, this buffer style is less likely to introduce latency into the VC datastream. The main reason for this is that the entire buffer does not need to be filled before the receiving protocol handler can begin processing the incoming packets. In the general sense, increasing the access granularity allows us to reduce some of the dynamic state overhead associated with a given buffer style, but we pay for that with an increased latency in the data stream. In theory, we could partially fill the buffer with valid packets and partially fill it with identifiably invalid packets. The receiving protocol handler would then be responsible for ignoring the invalid packets, but if the time taken to generate and filter out the invalid packets was less than the time it would take to fill the buffer with valid data, then we would reduce the overall delay. Even if this approach was tractable from an implementation perspective, we must still approach it cautiously as it increases the potential for variation of the latency in the datastream. We mentioned earlier, when discussing the communication

characteristics of the different VC models, how such jitter may be harmful in models such as the parallel harness.

7.2.2.3 FURI Protocol Handlers

FURI protocol handlers are the programmed tasks that execute within the framework of the FURI executive to implement a particular FURI protocol. The precise actions of a handler are specific to the protocol being implemented, but we can generalise its actions into two rôles: first, the protocol handler interacts with a given network channel to orchestrate the exchange of protocol packets and, second, the protocol handler processes the packet data it receives and, potentially, generates result packets. Essentially, each protocol handler provides an interface to a particular array resource or SLU. The exact feature set that a handler manages, though, is defined through the protocol itself. Effectively, it is the collection of protocol handlers currently active in the FURI executive that define the programmable logic interface. Furthermore, it is the ability to add and remove handler tasks through the FURI base protocol that facilitates the adaptive, packet based programmable logic interface we aspired to in Chapter 3.

We can differentiate protocol handlers according to how much they have been specialised to deal with a particular packet style or buffer format. If the protocol handler only ever processes packets of a fixed size and certain format then we can specialise its programmed implementation accordingly. For example, consider the implementation of a protocol handler that interacts with a FIFO buffer similar to that shown in Figure 7.10(ii), but containing fixed-size packets instead of the variable-sized packets shown. In this situation, the number of packets that must be consumed with each buffer access becomes static and can be folded into the code of the protocol handler. Each specialisation of this kind, where we fold detailed assumptions about the packet and buffer format into the protocol handler's implementation increases its efficiency at processing those packet and buffer types.

However, we have to trade off the specialisation applied to a handler against the constraining effect it has on which packets the handler can process. If the protocol handlers are too constrained, we may require more independent handler tasks running on the executive to handle the full diversity of the VC datastream. In particular, this means we effectively have to balance the handler's specialisation against the cost of context switching between many different handler tasks.

For example, in the sea of accelerators context, we could assert that each SLU is managed by a dedicated, specialised protocol handler. Since there might be a relatively large number of SLUs resident on the cell array in this VC model, there would be a correspondingly large number of handler tasks. Although the handler code is itself more efficient, the larger number of executive tasks means it may actually take longer before it is scheduled and the context switch penalty is more apparent. In this case, we can perhaps argue that the round-robin scheduling policy of the FURI executive should be replaced by an alternative policy that applies a higher level of reasoning in selecting which handler task should be executed next. A similar approach would apply to generalised protocol handlers that manage multiple input buffers where the protocol handler itself must make 'micro-scheduling' decisions about which of its multiple buffers to service next.

7.3 Implementing Virtual Circuitry Models

The discussion in the previous section addressed the general form and design space for FURI protocols and in this section we shall propose implementations of the three VC models in the FURI environment. The Data Encryption Standard(DES) is used as an example application in this section. Each implementation discussion proposes renditions of the DES algorithm in combinations of FURI managed circuitry and application program code to accentuate the rôle and challenges facing the FURI core in the different VC styles.

The DES is an appropriate algorithm to use in the proposed implementations

because its basic elements can be composed into the appropriate circuitry forms. For example, the DES contains many basic combinatorial computational elements which are individually appropriate candidates for becoming sea of accelerator SLUs. We can also explicitly wire all of the basic computational elements of the DES together within a pipelining parallel harness circuit. The resulting DES circuitry can then be presented to the FURI core for management. Specifically, its pipelined nature provokes the complex and particularly demanding timing issues that are characteristic of FURI managed parallel harness circuitry and that the FURI core must handle in parallel harness applications. The same computational elements can also be combined in a sequential algorithmic flexible harness that takes over the same fundamental rôle as the parallel harness interconnect circuitry but using highly optimised FURI code.

A short description of the DES is given in the following section.

7.3.1 The Data Encryption Standard(DES)

The DES is a 64 bit block cipher exploiting a 56 bit key length. It combines the two basic techniques of encryption: confusion (substitution) and diffusion (permutation). This section gives a very brief overview of the algorithm and a detailed description of the DES can be found in [91]. The algorithm, as shown in Figure 7.11, comprises 16 almost identical "rounds" which are bounded by initial and inverse initial plaintext permutations. The same algorithm is used for both encryption and decryption.

IP and IP^{-1} are the plaintext permutations applied at the periphery of the 16 DES rounds and they have very little cryptographic value. Software implementations of DES often omit them completely[9] as word-oriented microprocessors have difficulty implementing bit level permutations as efficiently as they can be done in circuitry.

[9] Although this means, strictly, that they no longer implement the DES and therefore typically refer to themselves as implementations of DES' or DES*.

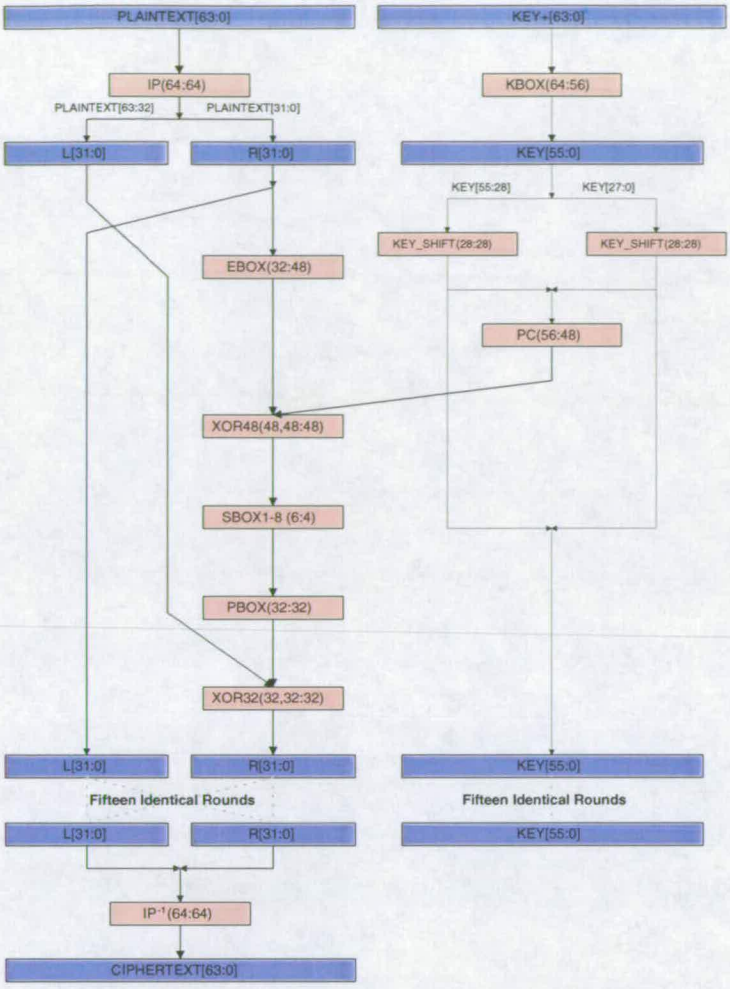

Figure 7.11: The Data Encryption Standard Algorithm

The Key Permutation(KBOX) is applied at the beginning of the encryption or decryption to extract the 56-bit key from the 64-bit input vector. Specifically, the permutation ignores the eighth bit of each byte which is typically used for parity. Key-shift is a circular shift of one half of the 56-bit compression key. For encryption, the 28-bit sub-key is shifted left according to a key schedule which varies how much the sub-key is shifted by, based on the round number. The situation is similar for decryption excepting that the sub-key is circularly shifted to the right and the key schedule re-creates the inverse key sequence to that applied during encryption. Permuted Choice(PC) is another bit-level permutation that selects 48 bits from the 56 bits contained in the shifted key. The resulting

value is then injected into the main flow of the algorithm to influence the encoding of the plaintext during the current round.

The expansion permutation(EBOX) is used to rapidly increase the dependency of every bit in the ciphertext on every bit in the plaintext. The permutation converts the 32-bit right hand data word to a 48-bit expanded data word by regularly repeating certain bits in the input word. Substitution Boxes(SBOX) are the main cryptographic feature of the DES and apply non-linear substitutions to 6-bit segments taken in sequence from the expanded data word after it has been XORed with the current round's encryption key. The resulting nibbles are re-packed into a 32-bit data word for the following permutation. The straight permutation(PBOX) is applied immediately after the SBOX substitutions and is essentially a standard permutation where no bits are repeated or omitted from the input word.

7.3.1.1 The DES modes

The basic DES algorithm is a symmetric block cipher where a ciphertext block is the direct product of the input plaintext and encryption key. In the standard algorithm itself, there is no cumulative relationship between plaintext blocks that are encrypted in sequence. This makes the encrypted ciphertext particularly susceptible to differential cryptanalysis techniques. To combat this, four DES operating modes are specified: electronic-codebook (just the application of the standard algorithm); cipher-block chaining; output-block feedback; and cipher-block feedback. Although the cryptographic aspects of these modes are interesting, the discussions below will consider just the standard, electronic-codebook operating mode.

7.3.2 The Application Context

The overall system context for this section was shown in Figure 7.1. Figure 7.12 expands that view to include more of the details from the discussions earlier in

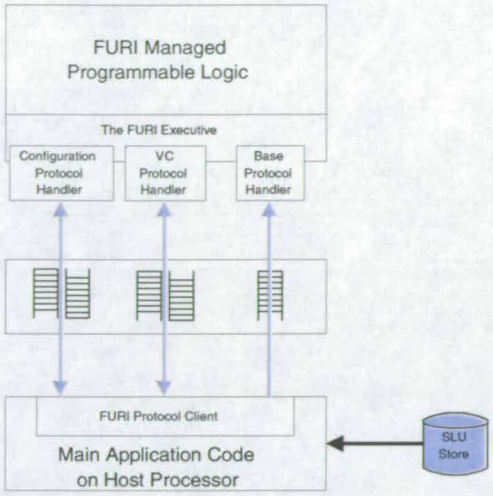

Figure 7.12: Lower level view of the FURI system context

this section and the discussion of the FURI base protocol given in Chapter 6. In all of the following examples, a FURI client executing on the host processor system allows the main application code to interact with the FURI managed programmable logic via a series of FURI protocols.

Initially, only the FURI base protocol is available and the FURI client must use it to instantiate other protocol handlers that are more specific to the VC model being supported. Figure 7.12 shows the state of the FURI managed programmable logic after the base protocol has been used to instantiate two handlers. One handler is dedicated to managing SLU configuration and the other to facilitating SLU interaction.

7.3.2.1 Configuration Protocols

Chapter 6 presented a detailed, low level discussion on the mechanics and issues related to the use of the FURI core in the rôle of configuration agent. The same fundamental issues raised in the discussion of a SLU loader in Chapter 6 also apply to the proposed VC model implementations discussed in this section. Since the discussion in Chapter 6 is fairly comprehensive, this section focuses more on the issues surrounding SLU interaction.

7.3.3 Sea of Accelerators

7.3.3.1 Overview

In the sea of accelerators implementation of the DES, the majority of the algorithm executes on the host microprocessor. We map one of the basic computations in the DES algorithm onto an SLU implementation. The main algorithm interacts with that SLU by passing packets of circuit operands to the appropriate protocol handler and collecting the corresponding result packets.

7.3.3.2 Interacting with a PBOX SLU

PBOX is an interesting candidate for a sea of accelerators SLU: the irregularity of the permutation makes it fairly difficult to implement efficiently in software, although an SLU implementation is nothing more than wiring between two 32-bit registers. Generally speaking, interacting with an SLU involves processing packets containing circuit operand data – the operands excised from that packet are presented to the appropriate SLU and the results generated by the SLU are captured and placed into a result packet for transmission back to the FURI client. Of the three VC models, SLUs in the sea of accelerators model have the simplest set of requirements associated with their interaction. Every operand packet received by the protocol handler for a sea of accelerators SLU would directly generate a corresponding result packet. This is in contrast with the more complex circuit state and timing issues that we must address for the Parallel Harness and Sequential Algorithmic models.

Interacting with the PBOX SLU is fairly straightforward as it has the simplest form of SLU interface – there is one input register, one output register, and both IO registers can be aligned to the same vertical positioning. All of these factors make the protocol handler's job of presenting operands and recovering results much simpler since the SLU registers share the same map register settings and, once the appropriate device state has been applied, they can be read and written

with a single FastMap transaction each. However, this is not the case for all SLUs and we must address the fact that different SLU interfaces place different requirements on the SLU interaction protocol handler.

7.3.3.3 Interacting with an EBOX SLU

The general form of the EBOX SLU is similar to that of the PBOX but with one important exception: the input and output registers of the EBOX are different sizes and, in particular, the output register exceeds the width of the FastMap data bus. We cannot rely on sharing map register settings and must now arrange for two separate FastMap reads to recover the result from the SLU's output. Comparatively speaking, interacting with the PBOX is much simpler than interacting with the EBOX solely because of the arrangement and form of its IO registers.

7.3.3.4 SLU Interface classifications

Figure 7.13 shows five common register interface arrangements for sea of accelerator SLUs and, underneath each, the approximate sequence of device state and interface register accesses required to effect a single computation. An underlying assertion in the first four interface arrangements is that the interface registers are at most 32 bits wide. The PBOX SLU we described above is an example of a Figure 7.13(i) SLU. In Figure 7.13(ii), the situation is slightly more complex since the input and output registers have differing sizes. This means that a different device state must be applied before accessing either to present the incoming operand or retrieve the corresponding result. Figure 7.13(iii) is an extension of the PBOX-style SLU demonstrating the effect of having multiple input registers in the SLU although, since the registers share common map register requirements, the only effect is an increase in the number of register writes corresponding to the number of input registers.

Figure 7.13(iv) shows three forms of SLU where there are multiple input registers, but without the ability to share map register state between accesses. In

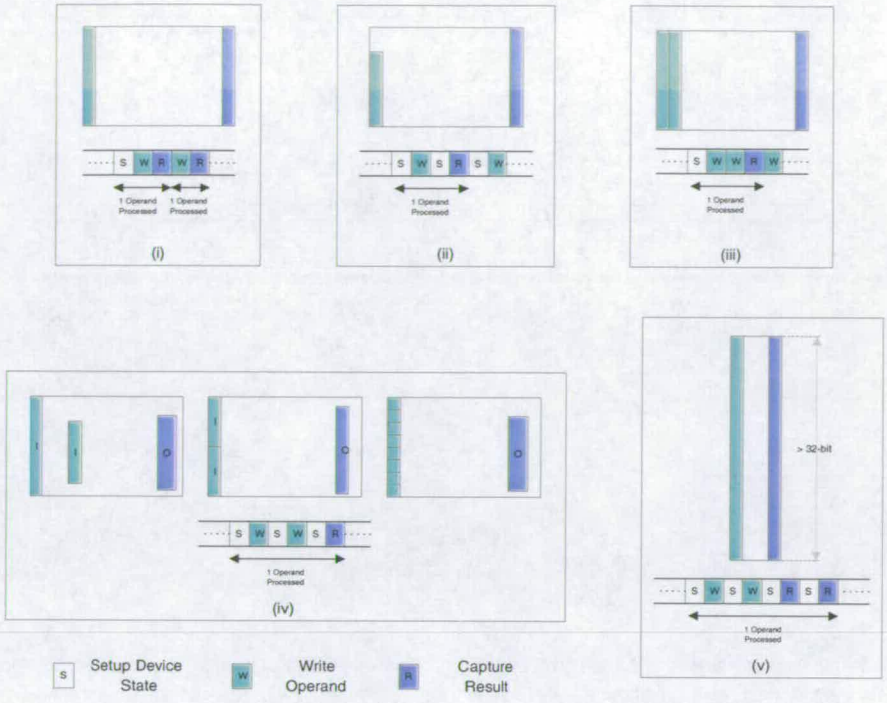

Figure 7.13: Interface Arrangements for FURI SLUs

this case, we must apply the appropriate control register settings before any read or write to an SLU register. In Figure 7.13(v), the 32-bit size constraint on the interface registers is removed so SLUs with a much wider data path can be considered. The EBOX SLU is a close relative of the Figure 7.13(v) since its output register is larger than 32 bits wide. Furthermore, although the tall SLU shown in Figure 7.13(v) is logically identical to the first interface type, the fact that it exceeds the width of the host FPGA's physical interface makes the presentation of operands and capture of results more complex. Presenting a single operand to one of the wide SLU interface registers requires multiple writes, each of which is preceded by an appropriate device state setting. A parallel situation holds for recovering an operand from a wide SLU output.

7.3.3.5 Managing interface arrangements in the Protocol Handler

The protocol handler requirements in the initial PBOX SLU description are fairly routine: the operand data in each packet can be neatly packed into the words

of the shared memory region and, with the appropriate placement constraints, the registers can share common map register values. However, the section above demonstrates that the amount of effort required to communicate with the SLU varies according to its interface style. Essentially, we must communicate formatting information to the protocol handler to let it know how individual operands are packed into the data payload of the packet. Furthermore, if the SLU has multiple input registers, we must somehow be able to address each operand to a particular destination register.

One immediate option is to specialise each protocol handler to a target SLU's interface. In particular, details of the operand characteristics, and transport scheduling are folded directly into the handler's packet processing code. At the opposite extreme, though, a single generalised protocol handler could be written to manage all possible SLU interface arrangements. To support each SLU interface arrangement, the packet's header information would contain all the necessary meta-data to allow the generic protocol handler to present the circuit operands correctly and, symmetrically, gather and re-pack the circuit results.

Generally speaking, neither of these situations is particularly ideal. We considered the effect of specialising protocol handlers within the sea of accelerators model earlier. The potentially large number of independent SLUs would increase the elapsed time between a given protocol handler being scheduled. The completely generic protocol handler would increase the control-data overhead in each operand packet. Furthermore, unless there are a large number of operands in each packet, the overall efficiency of the generalised handler would be relatively low because of its very general nature and correspondingly complex programming.

7.3.3.6 Dynamically binding to specialised SLU interface subroutines

One potential compromise between the two approaches, however, involves the late binding of operand processing to specialised interfacing routines that are dynamically generated when the SLU is first instantiated on the array. For example,

rather than just being responsible for transferring a SLUs bitstream image to the host FPGA's configuration memory, the configuration protocol handler can also create specialised instances of interface code templates. The specialised instances of the code templates essentially define two FURI subroutines. The first consumes operand data from a given position in the current packet and presents it to the specific SLU inputs whilst the other, when called, captures the output of the specific SLU and packs it into a given position in a results packet. Although there is a cost associated with generating the templates, we can consider this part of the overall configuration cost for the SLU and amortise it accordingly.

The important thing to note here is that we are not dynamically synthesising large amounts of code each time we process a new packet. Rather, the entry points for the synthesised interface code subroutines would be maintained in a binding table, keyed by an appropriately unique SLU instance identifier. Read and write operations to the SLU interface would be dynamically bound to the correct, specialised subroutines when we begin to process each operand packet. A crucial aspect of this style of protocol handler is an efficient dynamic binding scheme. The focused application of self-modifying code is very useful here – we can perform a single lookup of the interface binding table once per operand packet and overwrite the interface subroutine call entry points within the handler's code. This eliminates the cost of repeatedly calculating the dynamic binding at each SLU interface access and allows us to amortise the cost of the code self-modification as each operand is presented to the appropriate SLU. Conceivably we can amortise the binding cost over multiple operand packets, provided we can justify maintaining separate input FIFOs for each SLU.

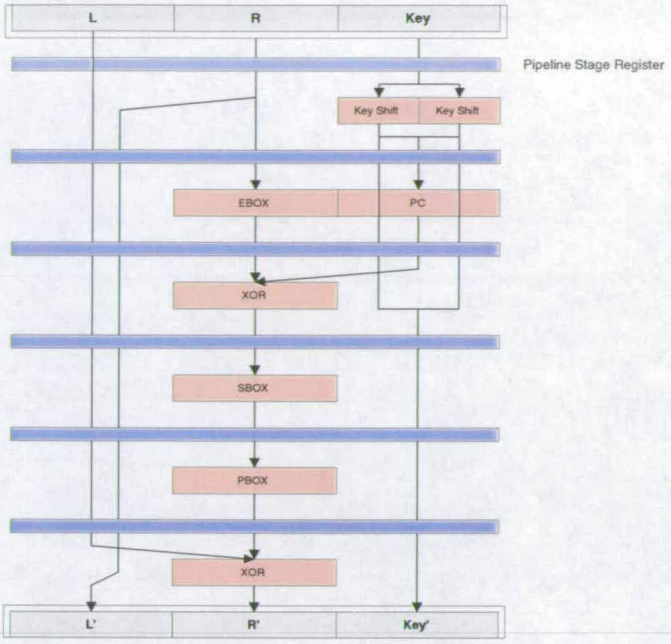

Figure 7.14: Pipelined Parallel Harness DES Circuitry

7.3.4 Parallel Harness

7.3.4.1 Overview

In the parallel harness implementation of the DES, we construct a single, large circuit that implements all the functionality of a single, highly pipelined DES round. As shown in Figure 7.14, pipeline registers of an appropriate width are placed between each of the main computational stages of the algorithm. For example, this would include pipeline stage registers between the EBOX and logic-XOR and the insertion of appropriate delay registers to ensure that the left word of the plaintext arrives at the round's final logic-XOR on schedule. We should note that, in this example, the harness itself is the pipeline register circuitry juxtaposed with the computational SLUs. As we described in Chapter 4, system SLUs are placed at the periphery of the harness circuit to form a register oriented wrapper interface. The parallel harness operand handler executing on the FURI core uses this wrapper interface to present operands and collect results from the circuit.

The intention here is not to propose a particularly fast or novel implementation of the DES circuitry, but rather to combine the basic computational elements of the algorithm in a parallel harness style circuit. The overall efficiency of the circuit itself is less important than the fact that it exhibits the timing issues that we must consider when implementing the parallel harness model on the FURI core. We should clarify that the pipelined timing of this circuit is fairly simple in comparison to other possible parallel circuit timing requirements. However, it is still appropriate for discussing the basic challenges that must be addressed within the circuit's protocol handler.

7.3.4.2 The challenges of interacting with a parallel harness circuit

The main challenge for the parallel harness DES circuit's protocol handler is that the presentation of an operand is temporally separated from the collection of its corresponding results. The mechanical aspects of presenting operands and capturing results is essentially the same as that used in the sea of accelerators: the interface arrangement influences the amount of work required to effect each interface access. However, the protocol handler must schedule its interface accesses to match the timing of the parallel harness circuitry. For example, when the parallel harness circuit is clocked from a freely-running clock source, it is imperative that the exact scheduling of SLU interface accesses coincides with the timing requirements of the harness circuit. If this is not the case, the protocol handler may miss results as they arrive at the circuitry output. There are three potential approaches to addressing this issue and each one is considered in turn below.

7.3.4.3 Matching the harness timing with specialised transport scheduling

The first approach requires a highly specialised and optimised implementation of the handler. Specifically, this handler uses advance knowledge of the DES harness's circuit latency and I/O schedule to synchronise the harness protocol

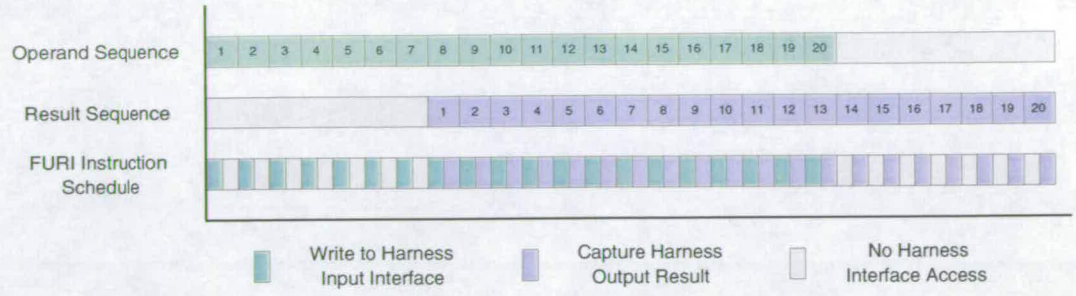

Figure 7.15: Operand and Result sequences for the pipelined DES Parallel Harness Circuit

handler's instruction sequence with the raw timing of the harness circuit. Specialised protocol handlers that are specific to a particular SLU are more likely in the parallel harness context because of the potentially low number of circuits sharing the spatial resources of the array at any one point in time. Scheduling an appropriate instruction sequence within the protocol handler is a task suited to the optimising stages of a higher level TTA compiler.

Aligning the execution of FURI instructions to the I/O schedule of the harness circuit is challenging and has two underlying assumptions. First, we assume that the FURI core and the parallel harness circuits are in physically compatible clock domains. The FURI core may exist in a separate clock domain from the circuitry it manages, for example, when the parallel harness circuitry has a longer critical path than the FURI core. Specifically, the clock frequency of the FastMap interface must be in phase with the clock being used at harness circuit's register interface. If this is not the case then we risk presenting an operand to the harness circuit at a point when it would not be latched. Alternatively, the FastMap interface may sample an incorrect result from the SLU output because its registers are in a transitional state.

The FURI core executes its instruction sequence serially but in this instance we need to capture results and present inputs in parallel. The core cannot perform both of these operations simultaneously in a single move instruction so the second

assumption is that we can actually execute enough FURI instructions within a single cycle of the DES harness circuit to present operands and capture results. Figure 7.15 shows a schedule of operands arriving at the inputs to the DES harness circuitry, and the corresponding result sequence. A breakdown of the interface accesses performed by the protocol handler is given underneath the operand and result schedules. We can see in this diagram that, after we overcome the initial latency of the DES harness's pipelined implementation, two interface accesses are required on each cycle of the harness circuit. However, sequential execution of FURI instructions means that the FURI core would have to execute more than one instruction within the clock period of the DES harness to give the illusion of capturing the harness's result and writing the next operand simultaneously. Effectively, the FURI core must have a very low instruction cycle time relative to the DES harness's cycle time. Supporting this practically is challenging since pipelining has the effect of reducing the clock period of the DES harness.

7.3.4.4 Isolating the FURI core timing from the DES harness

Whilst interacting with the DES harness at its natural timing is theoretically possible, we can see from the above discussion that a practical implementation would require a very fast implementation of the FURI core. The discussion in this section focuses on changing the style of system SLU at the harness circuitry's periphery as a means of making the task of interfacing with the circuit simpler. In both the cases presented, we increase the intelligence of the harness's interface SLUs to reduce the demands on the protocol handler and the FURI core.

Figure 7.16(i) shows the same DES harness circuit used in the discussion above, but this time bounded by system SLUs that contain FIFO circuitry. A particular point to note here is that the FIFO-based system SLU at the harness output is capable of stalling the harness circuit when the FIFO becomes full. This prevents the loss of results and, essentially, the use of FIFOs allows us to detach the internal timing of the harness from the FURI core. In this context, the FIFO

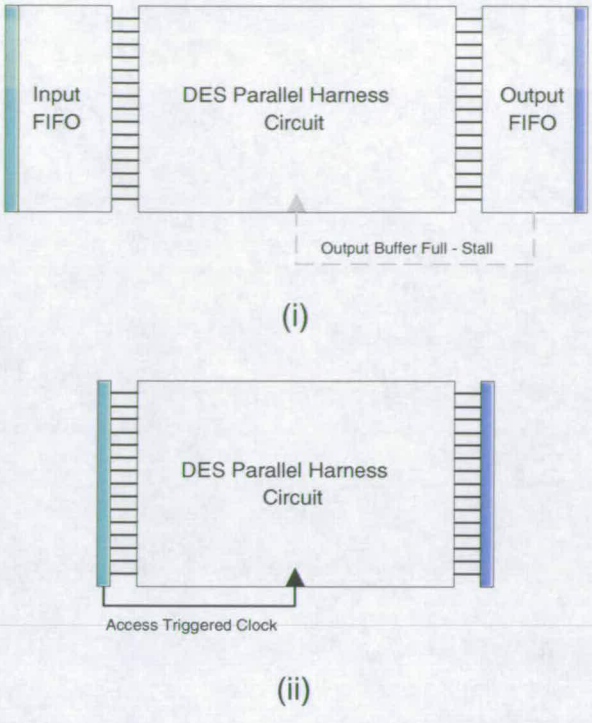

Figure 7.16: Parallel Harness DES circuitry

at the harness output allows the FURI core to interact with the harness circuitry at its own speed without risking the loss of any results. If we were dealing with a more complex parallel harness circuit which demanded new operands arrive at a particular rate, the FIFO at the harness input would also stall the harness circuit when it became empty, this time to protect any internal state that is required for the correct processing of future operands.

In the above approach, the FIFOs within the system interface SLUs provide a bridge across the two clock domains. The parallel harness circuit can still process operands and produce results at its core clock speed, even if the FURI core itself is not capable of supplying them at that rate. The third approach, shown in Figure 7.16(ii) uses a specific feature of the XC6200 to completely replace the harness circuitry's free running clock with a clock signal that is triggered once each time the FURI core captures a result or presents a new operand. Specifically, we use the FastMap interface's `regword` signal, described in Table 3.1, to replace the free

running clock with a single pulse that is generated each time a value is written to the DES harness's register inputs. In this context, the FURI core is actually implicitly responsible for clocking the DES harness each time it supplies a new set of operands. In this situation, the parallelism of the harness is still intact, but its computational rate is determined by the rate of its interactions with the FURI core.

An interesting point here is that the underlying XC6200 FPGA architecture gives us this facility essentially for free whereas, if we were to implement the earlier FIFO model, we would have to invest an explicit amount of circuit area. At the same time however, the implementation of FIFOs in an architecture such as Virtex is much more routine and less demanding of array resources. This is not to say that the two approaches are architecture dependent, though, since it is possible to implement FIFOs on the XC6200 and gated-clocks on the Virtex.

7.3.5 Sequential Algorithmic

7.3.5.1 Overview

We know from Chapter 4 that a sequential algorithmic circuit comprises a set of SLUs and FURI code that implements a flexible harness for interconnecting them. Figure 7.21 shows the general organisation of the sequential algorithmic DES implementation. Although the SLUs in this figure are in the sea of accelerators style, the SLUs used within the flexible harness could be either sea of accelerators style or parallel harness style. The programmed flexible harness combines them all into one conceptual circuit. From a comparative standpoint, the sequential algorithmic implementation of the DES used in this section is similar to the previous section's parallel harness circuit. Again, we are implementing a single DES round using SLU implementations of the main computational stages. However, the explicitly wired parallel harness interconnecting those elements is now replaced by the software routing executing on the FURI core.

7.3.5.2 Interacting with Sequential Algorithmic DES

The basic rôle of the protocol handler is unchanged in the sequential algorithmic model: it is still responsible for presenting operands to the harness circuit and collecting results. Like the parallel harness handler, the sequential algorithmic handler must also deal with the temporal separation of presenting operands and collecting the corresponding results. However the sequential algorithmic handler does not interact with circuitry configured on the array explicitly. Rather, it exchanges operands and results with the flexible harness task using shared memory FIFO queues as an inter-task communication mechanism.

There are two points we can note about this arrangement. The first is that, although the operand presentation and result gathering are temporally separate, because the protocol handler is not interacting directly with circuitry, we do not have the same complex timing issues that exist for the DES parallel harness[10]. The second point concerns the difference this arrangement makes to the measure of latency within the sequential algorithmic circuit. In the DES parallel harness, it is theoretically possible for the results from processing the current packet to be captured and re-transmitted within one scheduled run of the protocol handler. In this situation, however, the latency between an operand being presented and the result being captured depends on the flexible harness task being scheduled to run. The latency of the sequential algorithmic DES harness is therefore influenced by the scheduling policy implemented within the executive. The critical path of the sequential algorithmic DES circuit is defined by the runtime of the flexible harness code itself.

The flexible harness task is highly specialised to the particular form and layout of the SLUs it contains. This helps to counteract the loss of parallelism, incurred

[10]It is acknowledged that these timing issues could exist *within* a sequential algorithmic circuit if it contains a parallel harness SLU. However, the flexible harness task imposes a level of abstraction that that isolates the protocol handler from being directly concerned by the parallel harness timing issues. Instead, the timing issues are managed explicitly within the specialised flexible harness code.

as a result of the FURI core's serialised execution by giving the higher level TTA compiler the opportunity to deploy the optimisation techniques we discussed in Chapter 4. For example, because the location and form of the DES SLUs are fixed and the harness's communication sequence defined in advance, the compiler can apply operand multicasting using the XC6200's wildcards. The application of bin-packed writes using the XC6200's map register mechanism is also calculated at this stage. In the DES harness, for example, we can bin-pack multiples of the SBOX operands into a single write.

Essentially, the flexible harness code trades away as much of its dynamic decision making as possible to increase its efficiency. The downside to this, however, is that it limits the potential benefits available through some TTA optimisations. Operand sharing, for example, could be very effective if an operand becomes a semi-static value for a period of time. For example, whilst the plaintext inputs to the DES circuit are highly variable, the key itself may be static over successive operand processing cycles. We could potentially eliminate a series of data transports related to the key that are not required whilst it is in a semi-static state. An interesting way to implement this by analogy with the concept of run-time reconfigurable routing, discussed in Chapter 4, by dynamically patching the flexible harness's code. Instead of supplying new circuitry bitstream data to reconfigure explicitly wired routing, we actually supply new FURI program code and dynamically alter the flexible harness's software routing.

7.4 Performance Analysis and Projection

In this section we analyse the performance of the DES application implemented in the style of the three VC models.

7.4.1 Performance of the FURI core

Before we explore the implementations of the three VC DES implementations, we will first establish the instruction processing rates available through different configurations of the FURI core. In Chapter 5, we considered the FURI core's performance when loading SLU bitstreams at different clock speeds. In addition to scaling the clock period of the FURI core, a second dimension explored here is the effects of applying pipelining to the core's implementation. The overall aim of this section is to explore the instruction processing rates available through enhancements to the basic FURI core. By building cost models of the different VC DES implementations, and in combination with a cost model of the FURI Executive and its related components, we can determine the performance of the applications with respect to a given implementation of the FURI core.

Chapter 5 has a comprehensive description of the basic implementation of the FURI core. To recall, the basic implementation of the core required 19 cycles to execute each move instruction. Using this, we can derive an instruction processing performance relative to the core's clock speed. Furthermore, we can derive the instruction processing rate relative to different degrees of pipelining that we wish to apply to the core. These derivations are accurate indicators of the performance of each core implementation since each move instruction is executed in a deterministic manner and in constant time. Table 7.1 contains the instruction processing rates available at different clock speeds and pipelining configurations of the FURI core[11].

The viability of these performance details should be considered relative to a scaling of FPGA technology into future device generations. We can state that the application of pipelining to the core will involve increasing the gate count of the core's implementation. XC6200 technology, because of its relatively limited device

[11]The figures in Table 7.1 describe deeply pipelined FURI implementations down to cores supporing single-cycle instructions. We should note that such deeply pipelined implementations are tractable, provided we have a suitably high bandwidth memory hierarchy.

Clock Frequency	Basic, 19 Cycles per instr	16 Cycles per instr	8 Cycles per instr	4 Cycles per instr	2 Cycles per instr	1 Cycle per instr
8MHz	421052	500000	1000000	2000000	4000000	8000000
16MHz	842105	1000000	2000000	4000000	8000000	16000000
33MHz	1736842	2062500	4125000	8250000	16500000	33000000
66MHz	3473684	4125000	8250000	16500000	33000000	66000000
100MHz	5263158	6250000	12500000	25000000	50000000	1E+8
200MHz	10526313	12500000	25000000	50000000	1E+8	2E+8
400MHz	21052632	25000000	50000000	1E+8	2E+8	4E+8
800MHz	42105263	50000000	1E+8	2E+8	4E+8	8E+8
1GHz	52631579	62500000	1.25E+8	2.5E+8	5E+8	1E+9
1.2GHz	63157895	75000000	1.5E+8	3E+8	6E+8	1.2E+9

Table 7.1: FURI Core instruction processing rates at different clock speeds and with pipelining to reduce instruction cycle times.

density, may not be attractive for implementing a single cycle implementation of the core. A similar situation applies to the system clock speeds considered in the table. A 100MHz FURI core will tax the XC6200 technology we are targeting for the implementations of this thesis but circuitry speeds beyond 100MHz and into the 1GHz range are becoming tractable in the Virtex architecture and will be commonplace in future architectures.

We can assert that there is a relationship between the implementation technology and the degree to which we can pipeline and increase the clock speed of an implementation. As the target architectures gain density, the gate costs associated with pipelining become less relevant. Since pipelining enables higher clock frequencies above and beyond the frequency gains of the physical device architecture scaling, FURI instruction processing performance on the scale of billions of transports per second is conceivable.

7.4.2 Analysis of the framework costs and overheads

The data in Table 7.1 allows us to establish some basic instruction budgets according to a particular FURI implementation. In this section, we will characterise

the basic costs incurred by the fundamental elements of the FURI Framework. There are three framework components that we will examine for this purpose: the processing costs associated with the FURI executive and its basic system tasks; the costs for dynamically loading SLUs; and the costs for communicating new protocol handlers via the FURI base protocol.

7.4.2.1 Gathering Cost Information within the FURI Framework

Cost in this context relates to the number of move instructions that we must allocate to perform a particular task. In some cases, a static characterisation of the instruction cost is adequate. For example, we will present the instruction cost for the FURI executive's entry point code which captures the cost of initialising the overall FURI framework. This code is only executed once, when the FURI core first begins processing instructions and constitutes a static overhead that we must pay when bringing the FURI managed programmable logic online for the first time. Other tasks, such as loading an SLU, have costs that are relative to the context when the task is executing. The total instruction cost of loading an SLU depends on the size of the bitstream for that SLU and the parameters passed to the block loading subroutine. In this situation, the cost model used will have an assessment of any static instruction costs in the task and then a quantification of the dynamic cost involved in each unit of processing for that task. In the case of the SLU loader, the unit of processing we are interested in is, mainly, the cost of configuring each block of the SLU's configuration bitstream.

Throughout this section, the assessment of instruction processing costs comes from statistics generated by the FURI assembler on actual code. In detail, when the FURI assembler reaches the final stages of the assembly process, it can generate a breakdown of the number of actual move instructions associated with sections of the code being assembled. The assembler does not directly produce the dynamic characterisations discussed above but, with the static code costs available, we can calculate an appropriate dynamic model of the costs manually.

7.4.2.2 Costs associated with the FURI executive

As we discussed in Chapter 6, the FURI executive provides the low level runtime management of the software component of the FURI framework. The executive is the first layer of processing overhead within the framework that must be characterised. Because of its fundamental nature, however, the code of the executive has been optimised to increase its efficiency. This is reflected in the number of instructions that are used by the executive for basic system tasks such as making the decision on which task should be the next to be scheduled.

Table 7.2 shows the main code costs incurred by the FURI executive. The startup costs incurred when the FURI core first begins processing are 1775 move instructions and the recurring cost each time the executive selects a new task to run equates to 72 move instructions. As part of its setup process, the FURI core populates its task list with two basic system tasks: the first is an "idle" task that costs 130 instructions each time it is scheduled; and second is the FURI base protocol task. Quantifying the costs associated with the base protocol is slightly more complex than the previous components because of the nature of the protocol itself. Each base protocol packet contains a mixture of code and data and each time the protocol handler is scheduled to run, it checks the base protocol's packet buffer for a new packet. If one is found, the code section within the packet is executed.

The protocol handler itself has a packet processing overhead of 92 instructions. The cost of executing the code section of each packet must be added to this to determine the cost of communicating any new protocol handlers. The packet buffer for the base protocol, as it was implemented for this thesis, is 1024 words in length. A utility written to assist FURI clients executing on a standard PC architecture generates base protocol packets that meet the processing conventions of the base protocol handler. The packets generated are structured such that the

Code Section	Description	Instruction Cost
System Entry Point	First code sequence executed by the FURI core on power-up. Performs general housekeeping and triggers loading of the kernel SLUs	14
Loading Kernel SLUS	High speed transfer of kernel SLU bitstreams to host FPGA's configuration RAM	1385
Subroutine Initialise	Housekeeping tasks associated with preparing the subroutine call stack	2
Executive Initialisation	Harness code orchestrating the setup of the Executive	18
Task-list Setup	Preloading of the task list data structure to reduce recurring scheduler overheads	222
Initial Task Loading	Repeated calls to the `add_task` subroutine to populate the executive's task list with the basic system tasks (idle task and base protocol)	62
Add Task	Actual cost of inserting a task into the task list.	31
Remove Task	Cost of ending a task's execution permanently by removing it from the task list completely	48
Executive Main Loop	Selects a task from the task list and runs it. The only recurring cost in the executive code after the setup phase has passed. Every task switch has this as an overhead.	72

Table 7.2: Breakdown of the instruction costs for the FURI Executive

Code Section	Description	Instruction Cost
Recurring Packet Processing	Costs involved in managing the exchange and processing of the configuration packets and general buffer management	97
Static protocol overhead	Housekeeping performed at the end of the configuration protocol	62
Code Buffer Generation	Cost of dynamically generating instructions that perform the transfer of configuration data. The exact cost for the code buffer is the product of this figure and the code block size.	$94 \times block\_size$
Code Buffer Exit	Static cost incurred after each code block is executed	100
Executing Code Buffer	Cost of executing the code block. The exact cost is scaled according to the chosen block size.	$block\_size$

Table 7.3: Breakdown of the instruction costs from the configuration protocol.

data payload is one third of the overall packet size[12]. The remaining two thirds of the packet contents, equivalent to 680 words (340 instructions), contain the code executed each time the base protocol processes a packet. The setup costs of each of the VC DES implementations below include the time taken to transmit and process the base protocol packets that make the handler code resident within the FURI core.

The last system component considered in this section is the configuration protocol handler. To recall, the configuration protocol handler is responsible for instantiating SLU bitstreams communicated over the FURI network environment. Packets of configuration data arrive and the protocol handler uses the block-based configuration subroutine from Chapter 6 to transport the bitstream data from the packet payload to the host FPGA's configuration memory. A breakdown of the costs associated with this protocol handler is given in Table 7.3. The actual cost for loading a particular bitstream depends on the chosen block size,

[12]We should recall that, for efficiency reasons, the base protocol buffer contains only a single packet which is consumed and overwritten in its entirety. The packet size mentioned here is, therefore, equivalent to the size of the base protocol buffer itself.

the number of configuration blocks within any given packet payload, and the number of packets required to transmit the SLU's bitstream data. From our earlier experiments in Chapter 6, we saw that the block loader's efficiency peaked when size of the bitstream blocks being configured was approximately 128 words long. An important thing to note from Table 7.3 is that generating the code buffer for each block is an expensive operation due to the calculations required to synthesise instructions. The number of blocks we process per packet depends on the size of the packet payload. The working figure chosen for packet payload size is 512 data words although, if we were exploring different ways to optimise the loader protocol, adapting this figure dynamically to amortise protocol processing costs would be the first optimisation.

Overall, the cost of loading a bitstream, taking into account some of the main costs from Table 7.3, is:

$$packet\_processing = block\_count \times ((94 \times block\_size) + 100 + block\_size)$$

$$loader\_processing = (packet\_count \times (packet\_processing + 97)) + 62$$

We can use these expressions to calculate the loading costs associated with the DES SLUs used in the VC DES applications below. Table 7.4 contains the instruction costs associated with loading each of the DES SLUs used in the three VC DES implementations. We can consider briefly the loading times for some of the SLUs in this Table, relative to some of the key FURI configurations of Table 7.1. The PBOX SLU, on a 33MHz basic FURI implementation as a loading time of approximately 0.72 seconds. The parallel harness SLU on the much faster, 1.2GHz single-cycle FURI implementation has a loading time of just over 1 second. Both of these timings are relatively slow as a result of the processing overheads associated with the block loader and its code buffer. We shall return in the later part of this section to consider enhancements that would reduce the SLU loading costs and timings. Also, as mentioned above, we assume a packet

SLU Name	Size (words)	Transmitted Packets	Loading Cost
EBOX	3844	8	3237662
PBOX	2352	5	1264922
KBOX	5316	11	6120770
Key Shift	1356	3	455522
Permuted Choice	3500	7	2478930
SBOX1-8	8086	16	12948990
Parallel Harness DES	86876	170	1.46E+9
Sequential Algorithmic DES	81060	159	1.28+9

Table 7.4: Configuration costs for the DES examples.

payload of 512 words and, in this instance, we assume the configuration protocol handler has a communications buffer capable of holding in the order of 6 to 8 unprocessed packets.

7.4.3 VC DES Implementations

In this section we discuss details of three VC DES implementations with the aim of deriving instruction costs for each implementation from which we will ultimately make performance assessments. Where relevant, we will characterise three costs for each of the implementations: an implementation setup cost; a maintainence cost; and an active processing cost. The setup cost encapsulates the overheads associated with transferring any protocol handlers and SLUs before application processing begins and the maintainence cost characterises the instruction costs incurred by the application when it is scheduled, but has no outstanding application processing to perform. The active processing cost captures the instruction costs for each implementation when it is actively processing packets, implementing the given VC DES model.

7.4.3.1 Sea of Accelerators VC DES Implementation

In this section, we will consider a basic implementation of sea of accelerators style DES. The organisation of this implementation is shown in Figure 7.17. In

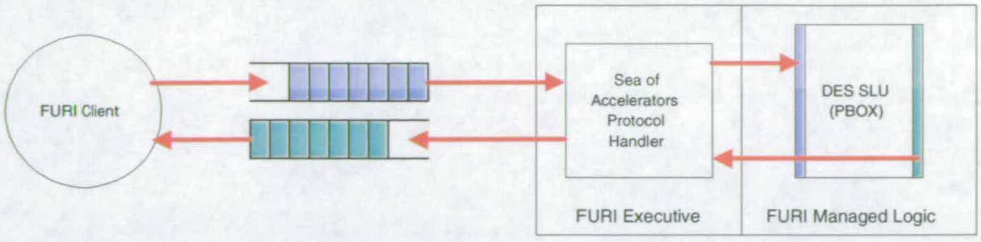

Figure 7.17: Basic Sea of Accelerators VC DES Implementation

detail, we are concerned with instantiating and interacting with only a single sea of accelerators style SLU through a dedicated protocol handler.

In the setup phase, the FURI client uses the base and configuration protocols to transmit both the protocol handler and the SLU to be managed. From that point, the application protocol handler is then ready to process operand packets. The SLU we are managing for this basic implementation is the DES PBOX SLU. We know from the previous discussion that this has an instruction budget requirement of 1776496 instructions. Based on the assembly statistics of the code implementation, the total size of the protocol handler is 161 instructions. This fits within a single base protocol packet and so we can assert a base protocol transmission cost of 432 instructions. A breakdown of the costs for the protocol handler is given in Table 7.5.

Establishing the maintainence and active processing costs of the handler requires a breakdown of the total handler cost into sections identified for protocol overhead versus operand processing. Again, from statistics generated by the FURI assembler from the protocol handler's code, the handler consumes 85 instructions when there are no operand packets to process. The active processing cost of the implementation is 33 instructions per operand, but the cost of actually processing each packet is relative to the operand payload of the packet and the protocol overheads. From instruction costs generated by the FURI assembler, we can characterise the active cost of this implementation as:

Code Block	Description	Instruction Cost
Protocol Overhead	Instruction costs from managing the packet buffer and integration of the handler with the FURI Executive	85
Packet Overhead	Instruction budget required to process packet headers and prepare the protocol handler for dealing with given input and output packets	43
Operand Processing	Instruction budget for applying an operand to the SLU inputs and capturing its output into the result packet. The packet sequence IDs that are used to allow the client to relate result packets to operand packets are preserved as part of the packet overhead.	33

Table 7.5: Breakdown of costs for the sea of accelerators VC DES protocol handler.

$$soades\_active = (((33 \times packet\_payload) + 43) \times packet\_count) + 85$$

The exact figure chosen for the packet payload component of the above expression is generally application dependent but the figure used in this discussion is 64 words. This is chosen to balance the protocol cost and operand processing cost to give a packet processing overhead below 5% of the protocol handler's instruction budget. From our earlier discussions characterising the likely payloads within a sea of accelerators application, we know that we cannot assume large packet payloads: the PBOX SLU is likely to be shared with multiple applications (one example of this would be, multiple encryption sessions within the network stack of a host system) and preventing latency crosstalk between the FURI communication streams advocates compact operand payloads. Actively processing a packet requires each operand is presented to the SLU from the source packet's payload section after which the corresponding result is captured from the SLU's output and placed in the payload of a result packet. Given all these considera-

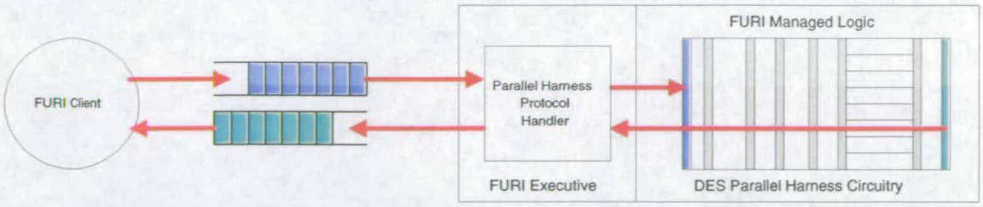

Figure 7.18: Parallel Harness VC DES Implementation

tions, we can assert that the overall cost for processing each operand packet in this implementation is 2155 instructions.

The PBOX SLU, as we have noted previously, is the most benevolent of the DES SLUs for transporting operands to and capturing results from. We can explore, briefly, the cost differences that would occur if we were to interact with a different DES SLU. If we replace the PBOX SLU with the EBOX SLU, for example, we must now manage extra device state context to transport operands to registers in the SLU interface that can no longer share device contexts. This results in an increase in the operand processing budget of the order of 5 instructions. In this case, we must invest an extra 320 instructions to process the same data packet used with the PBOX SLU.

7.4.3.2 Parallel Harness VC DES Implementation

The overall organisation of the parallel harness implementation of VC DES, shown in Figure 7.18, shares some of the basic features of the sea of accelerators implementation. Again we have a single protocol handler interfacing to clients on the FURI network that are transmitting operand packets for processing within the parallel harness style DES SLU. The main difference in this implementation is that we must manage the temporal aspect of the SLU's behaviour. The parallel harness SLU used here is based on the pipelined DES circuitry shown structurally in Figure 7.14. The algorithm is fully unrolled and pipelined with 6 stages per round. The circuitry is not freely clocked – it exploits the transport triggered clocking mechanism described in Section 7.3.4.4. Each time we write to the SLU's

register that holds the last byte of plaintext entering the harness, it triggers a clock pulse within the harness.

Effectively, the harness circuitry manages its own timing and has an input interface comprising two 64 bit registers: one to hold the encryption key applied to the plaintext block and the other to hold the plaintext block. The output of the harness is the 64 bit ciphertext register. This output is not buffered and an unread result will be lost if not read before the next harness clock cycle is triggered. In total, sixteen 6-cycle DES rounds within the harness result in a 96 cycle latency within the harness. Before the protocol handler begins to read any results from the harness, it must first issue 96 operands. After this point, the handler may issue and capture operands until all packet payloads have been exhausted. At this point, if no new packets arrive, the handler flushes the SLU to capture the 96 results delayed by the circuit latency. The breakdown of the instruction costs for this implementation is given in Table 7.6.

The encryption key is static for each operand packet and the first two words of the packet payload specify the key for the plaintext operands contained in the remainder of the payload. This forms part of the packet processing overhead and remains essentially independent of the overall operand processing cost. Because of the nature of the model and the resulting latency within the harness circuitry, the packet payloads communicated to the implementation are large, on the scale of 512 operands per packet. This guarantees, that we will not repeatedly incur a penalty for flushing the harness, especially when we consider the likely availability of multiple protocol packets.

The setup cost for this application is quite large although this mainly comes from the costs associated with loading the harness SLU itself. We will address this as a key performance issue in the Performance Projections section later in this chapter. The protocol handler for VC DES is 210 instructions in length. This fits within a single base protocol transmission packet and, like the sea of

Code Block	Description	Instruction Cost
Protocol Overhead	Instruction costs from managing the packet buffer and integration of the handler with the FURI Executive	98
Packet Overhead	Instruction budget required to process packet headers and prepare the protocol handler for dealing with given input and output packets and setup the harness SLU with the encryption key from the packet payload	60
Harness Write	Transport of a single operand to the plaintext input register of the harness.	24
Harness Read	Transport of a single result from the ciphertext output register of the harness.	24
Harness Write-Read	Combined transport and capture of operand and result, respectively, to the harness SLU. Includes a quantification of a loop control overhead incurred at this stage.	52
Harness Populate, Harness Depopulate	Instruction cost to overcome the pipelining latency of the harness SLU. This cost is incurred when we populate the harness and again when we depopulate the harness.	2304

Table 7.6: Breakdown of costs for the Parallel Harness VC DES protocol handler.

accelerators implementation above, our overall base protocol transmission costs 432 instructions. From our earlier table of loader costs, we can see that the parallel harness DES SLU is extremely expensive at 1.36E+9 instructions.

The maintainence cost for the handler, at 98 instructions, is comparable to the maintainence cost of the sea of accelerators handler. However, the cost is generally less applicable here because we know from our earlier discussions that we are less likely to share the array with multiple parallel harness SLUs. Conversely, the maintainence cost for the sea of accelerators is important for exploring the saturation point of the FURI core when multiple protocol harnesses reside within the executive.

We have approached the active processing cost of this handler in the earlier discussion on the latency characteristics of the harness SLU. There are three distinct phases the handler goes through when processing an operand packet. If this is the first packet since the handler has been re-scheduled, the harness is populated with the first 96 operands from the first packet's payload at an instruction cost of 2304. After this threshold has been met, we must iteratively apply and capture operands and results until the packet payload has been exhausted. This costs 52 instructions on each iteration, during which we are effectively processing one operand completely. This iterative harness write-read continues until all of the packet data has been consumed. The buffer in this implementation operates circularly to circumvent pipeline stalls: new packets may arrive from the FURI clients as packets are being processed and consumed[13]. When the datastream itself stalls, the pipelined SLU is flushed by the handler to capture the outstanding results from operands input 96 cycles earlier. Again, this comes at a total cost of exactly 2304 instructions. From this, we can assert that the cost for processing one 512 word packet is 26398 instructions, giving an average cost of approximately

[13]The shared memory synchronisation facilities of the detacher task facilitate this, but we do not consider the detacher cost in these performance assessments since it is a specific feature of the implementation within the test platform. Its resource requirements are normalised out of these performance assessments.

51 instructions per operand. The general cost of this handler, if we prolong the datastream, is:

$$active\_cost = (2304 \times 2) + ((datastream\_length - 96) \times 52) + static\_overheads$$

We should note here that the first 96 operands processed do not pay the overhead for loop control that iterated operands do[14]. Effectively, this means that the first 96 operands of this implementation have a better average processing cost (48 instructions per operand) than iterated operands (52 instructions per operand). This difference is entirely due to the unrolling of loops that would have otherwise iterated through the population and depopulation of the harness circuitry. If we pay the spatial costs and unroll the code of the whole payload processing region of the harness then we would obtain, at best, an average raw operand processing cost of 48 instructions.

7.4.3.3 Sequential Algorithmic VC DES Implementation

The organisation of the VC DES sequential algorithmic implementation is shown in Figure 7.19. The primary difference between this implementation and the two we have explored above is that the protocol handler is split into two components: the first of these interfaces directly the FURI network packet buffers and, by extension, the FURI clients; and the second implements an optimised series of transports through 13 DES SLUs (EBOX, PBOX, KBOX, Key Shift, Permuted Choice, the eight SBOX SLUs, and a wide XOR) instantiated for the implementation.

The main focus of this implementation discussion falls on the costs associated with the second handler component. The first component essentially performs

[14] We should clarify that the 2304 cost is applied twice because it is representing the initial population of the harness SLU and the final depopulation of the harness when we have no more operands to process. Each investment of 2304 actually covers half the cost of completely processing an operand.

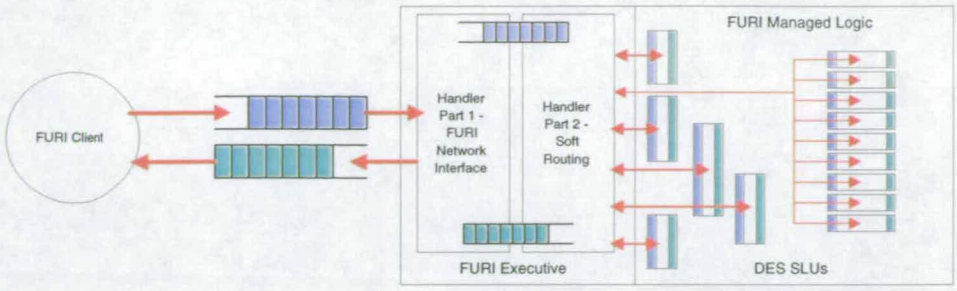

Figure 7.19: Sequential Algorithmic VC DES

very basic packet handling and has an overhead equivalent to the protocol and packet processing overheads of the parallel harness handler. However, there are no equivalent costs to operand processing since there are no direct circuit interactions.

The sequence of 14 data transports performed in the soft routing handler to implement a single DES round are specified in Figure 7.20. The instruction cost associated with actually implementing this sequence is 48 instructions. Transforming the operand into the completed DES ciphertext requires 16 such rounds so we can declare the processing cost for each operand as 768 instructions. This is the primary recurring cost for the model implementation and the overriding processing cost since it applies to the processing of each operand. There were two optimisations applied to reduce the amount of processing within the soft routing code. First, a suitable floor planning for the SLU circuitry was devised to limit the number of device context transitions required whenever a transport was being made. For example, we may align the interfaces of SLUs so that they all begin at the same row in the host array. This gives a degree of commonality to the device state that must be applied to access each SLU's interface. The second optimisation involved packing multiple operands into a single data transport. This allows us to target the SBOX SLUs and, rather than performing 8 individual data transports, we make a single transport that affects all 8 SLUs because their interfaces have been vertically aligned. If the operands are packed in the correct

$$
\begin{aligned}
R &\rightarrow EBOX\_in \\
EBOX_o ut &\rightarrow XOR\_in1 \\
key[27:0] &\rightarrow KeyShift\_in \\
KeyShift\_out &\rightarrow key[27:0] \\
key[55:28] &\rightarrow KeyShift\_in \\
KeyShift\_out &\rightarrow key[55:28] \\
key &\rightarrow PermutedChoice\_in \\
PermutedChoice\_out &\rightarrow XOR\_in2 \\
XOR\_out &\rightarrow SBOXES\_in \\
SBOXES\_out &\rightarrow PBOX\_in \\
PBOX\_out &\rightarrow XOR\_in1 \\
L &\rightarrow XOR\_in2 \\
R &\rightarrow L \\
XOR\_out &\rightarrow R
\end{aligned}
$$

Figure 7.20: Data transport sequence applied in a single round of the Sequential Algorithmic VC DES Implementation

bit positions within the word being transported, it will arrive at the correct SLU's input.

In the wider context, the packet payloads that are relevant to the sequential algorithmic implementation required some consideration. The separated architecture of the protocol handler is designed to prevent the network processing side of the implementation from being stalled by the relatively slow soft routing handler. We discussed earlier how the communications characteristics of the sequential algorithmic model resembled that of the parallel harness, but without the same degree of complex timing requirements. Whilst these requirements drove us to provide large packet payloads in the parallel harness, for the sequential algorithmic model we can return to more compact payloads. In this implementation, we assert that the packet payload in each packet is in the order of 128 operands and has a processing cost in the order of 100165 instructions. The buffers interconnecting the two protocol elements handle "bursty" datastream traffic. The

network buffers, on the other hand, do not need to be deep because the responsive network handler consumes packets rapidly, populating the internal reservoir of outstanding operands and depopulating the set of outstanding results. For this implementation, network buffers of depth 5 would provide sufficient numbers of operands to keep the soft routing handler active.

The setup costs for this implementation, like the costs we have seen in the Parallel Harness implementation, are high as a result of the amount of SLU loading that occurs prior to operand processing. For this implementation, we configure the FURI managed reconfigurable logic with the bitstream information for 13 DES SLUs. In total we require in order of 1.19 billion instructions, a comparable figure to the parallel harness implementation and significantly high. The increased size of the protocol handler code required for this model, means that we now require the transmission of two base protocol packets at a cost of 864 instructions. Clearly, the base protocol costs are dwarfed by the configuration overheads for the implementation's SLUs and the overall setup cost is best characterised through the configuration costs of the SLUs.

Because of the relative simplicity of the protocol code surrounding the soft routing core of the model implementation, the packet and protocol processing overheads are not particularly interesting for this model. The above discussion on operand processing costs gives a sufficient characterisation to carry into the discussion and performance projections in the following section.

7.4.4 Performance Projections

Table 7.7 collates the salient features of the three VC DES implementations described above. Our aim in this section is to take the analyses from the previous sections and derive some application oriented performance assessments for different configurations of the FURI core. We will relate these results to some of the key implementations of the DES algorithm on other architectures and technologies.

Model	Setup cost	Maintainence cost	Processing cost (per operand)
Sea of Accelerators	2352992	85	33
Parallel Harness	2.72E+9	98	48–52
Sequential Algorithmic	2.38E+9	100–150	768

Table 7.7: Summary of the main instruction costs from the three VC DES implementations

Additionally, we will consider how the implementations can be enhanced to project the performance of the FURI framework onto future generations of device architecture. By retrospectively analysing the implementations we can identify their main performance limiting features and explore how modifications to the approach and environment would affect performance.

We will begin our performance assessment by deriving the number of operands per second we can process for the sea of accelerators implementation. We are assuming the protocol handler has packet buffers capable of holding 5 operand and result packets respectively. In this situation, the protocol handler has a workload of 10575 instructions to completely process the incoming packet buffer before the executive schedules other tasks, incurring a 72 instruction switching cost. We assume here that the FURI core is lightly loaded, essentially allowing the protocol handler to be rescheduled immediately after it finishes processing one buffer set. Based on the workload above, the instruction processing rates from Table 7.1, and assuming that the model's setup phase has passed, Table 7.8 contains the sea of accelerator operand processing rates on a selection of FURI core implementations.

Tables 7.9 and 7.10 give the processing performance of the parallel harness and sequential algorithmic implementations respectively. The same approach used for the sea of accelerators is applied to both models when generating these tables, with the main difference in each instance being the seeding of the workload to packet ratio. This ratio is tailored in each table to follow the packet payload

Core	Packets/Sec	Operands/Sec	Mbit/sec
19 Cycle, 33MHz	822	52608	1.6
19 Cycle, 100MHz	2465	157760	4.8
19 Cycle, 1GHz	24885	1592640	48.6
8 Cycle, 33MHz	1951	124864	3.8
8 Cycle, 100MHz	5911	378304	11.5
8 Cycle, 1GHz	47282	3026408	92.3
2 Cycle, 100MHz	23641	1513204	46.2
2 Cycle, 400MHz	94563	6052032	184.6
2 Cycle, 1GHz	236407	15130048	461.7
1 Cycle, 1GHz	567376	36312064	1108.1

Table 7.8: Processing Performance of Sea of Accelerators VC DES

Core	Packets/Sec	Operands/Sec	Mbit/sec
19 Cycle, 33MHz	66	33792	1
19 Cycle, 100MHz	198	101376	3
19 Cycle, 1GHz	1994	1020928	31.1
8 Cycle, 33MHz	157	80384	2.4
8 Cycle, 100MHz	474	242688	7.4
8 Cycle, 1GHz	3789	1939968	59.2
2 Cycle, 100MHz	1895	970240	29.6
2 Cycle, 400MHz	7577	3879424	118.3
2 Cycle, 1GHz	18941	9697792	295.9
1 Cycle, 1GHz	45458	23274496	710.2

Table 7.9: Processing Performance of Parallel Harness VC DES

descriptions given earlier for each of the implementations. For the parallel harness model, we consider the protocol handler processing a single packet with a 512 word payload. To process this, the FURI core must process 26398 instructions. For the sequential algorithmic model, we consider processing a single packet with a 128 operand payload and a workload requirement of 100165 instructions per packet.

7.4.4.1 Comparing the three VC model implementations

In this section we present some conclusions on how the three VC models relate to each other based on the results we have presented. The first statement that we can make is that, strictly in terms of per operand processing costs, the sea of

Core	Packets/Sec	Operands/Sec	Mbit/sec
19 Cycle, 33MHz	18	2304	0.07
19 Cycle, 100MHz	53	6784	0.2
19 Cycle, 1GHz	527	67456	2.05
8 Cycle, 33MHz	42	5376	0.16
8 Cycle, 100MHz	125	16000	0.48
8 Cycle, 1GHz	1000	128000	3.9
2 Cycle, 100MHz	500	64000	1.95
2 Cycle, 400MHz	2000	256000	7.81
2 Cycle, 1GHz	5000	640000	19.53
1 Cycle, 1GHz	12000	1536000	46.8

Table 7.10: Processing Performance of Sequential Algorithmic VC DES

accelerators was the most effective model, followed by the parallel harness and then the sequential algorithmic model. This is a valid conclusion, even although the ranking of the models is just what we would expect based on the level of prescribed level of effort required to process operands.

Setup costs are very expensive in all three models relative to the actual processing costs within the models. This is predominantly due to the inefficient nature of the kernel of the configuration protocol that has been implemented. However, this does not affect our overall ranking of the models in terms of their effectiveness: the operand processing costs are more relevant over time as the cost of loading SLUs is amortised over successive operands. We return to tackle the poor loading performance in a later section, describing some specific performance enhancements.

Beyond these basic conclusions, it is difficult to make further comparisons between the three models that are not biased by the model's intrinsic suitability for the particular application. We must acknowledge that the models have different aspects that will suit a given application to different degrees. In relation to this, we can say that the sequential algorithmic implementation of DES has, overall, very poor processing performance with respect to the other two model implementations presented here. We should interpret this as signifying that the

many hundreds of data transports required to implement the soft routing harness and process a single DES operand goes beyond what is sensible for a sequential algorithmic VC application. Using same model to interconnect a few SLUs of coarser granularity would provide more promising results for the model in terms of performance.

7.4.4.2 Comparison to existing implementations

In this section, we attempt to compare the performance of existing DES implementations to those we have described in this chapter. Table 7.11 quotes the DES performance statistics collated in [85].

Of the three model implementations, only the performance results of the parallel harness and sequential algorithmic implementations are directly comparable to the performance figures quoted in Table 7.11. In the Sea of Accelerators implementation we are only partially completing the DES algorithm. We must bear this in mind, even although we can see that the raw performance details are higher for this model than any of the others.

We can roughly extrapolate a performance estimate for a full DES implementation in the sea of accelerators style if we consider explicitly replacing the single PBOX SLU with a sea of accelerators SLU that implemented the entire DES algorithm. We differentiate this organisation from the parallel harness implementation, based on the different timing models used in the SLUs. We treat the timing of the parallel harness SLU explicitly whereas the sea of accelerators SLU, in its pure form, is expected to be internally stateless and therefore have no timing issues to manage. For an implementation of a full sea of accelerators DES, the operand processing costs would increase slightly to reflect the increased complexity of the SLU interface (from a 32 bit input to a 128 bit input, and from a 32 bit output to a 64 bit output). The performance results we observed for the earlier PBOX sea of accelerators implementation should map closely onto the full DES implementation. There are two points we should note however: the

first is that the raised operand processing cost from the increased SLU interface complexity means that the processing rate for the full sea of accelerators DES has a lower bound equivalent to the performance from the earlier parallel harness VC DES implementation; and, second, the setup costs associated with the sea of accelerators would increase to be directly comparable with the parallel harness and sequential algorithmic costs. We will still amortise the setup cost more rapidly in the sea of accelerators full DES than in either of the other two models. However, because the setup costs are now much more comparable, we can state that the actual points at which the setup cost becomes negligible for each model will reflect, proportionally, the operand processing costs of the three models.

For the two complete algorithm implementations, the performance as quoted for these basic implementations lags behind what has been achieved in other technologies and architectures. The parallel harness comes closest to gigabit computational rates, with a throughput of over 700Mbit/sec on the fastest FURI configuration. This is still slower than the contemporary implementations that approach 10Gbit/sec rates with lower clock frequencies. One point that we should note about the higher performance implementations, however, is that they are reflections on the core speed of the DES circuitry rather than performance results taken from directly equivalent VC style implementations. The system interfaces for these high performance circuits come in the form of dedicated communications ports tied to actual device pins. None of the cases directly consider the VC style of SLU interfacing that we have explored in our implementations.

7.4.4.3 Performance enhancements and projections

From the performance comparisons above, we have seen that a significant clock speed and pipelined implementation of the FURI core is required before the performance of the VC DES implementations approach existing circuit implementations. In this section we will propose some enhancements to the implementations and consider modifications to the device architecture to better support our VC

Year	Technology	Clock Rate MHz	Throughput Mbit/sec	Unrolling
1997	Xilinx 4000	7	26	none
1998	Xilinx 6000	23	57	none
1999	Xilinx 4000	43	172	none
2000	Xilinx S2	94	376	none
1998	Xilinx 4000	25	384	partial
1998	Virtex	101	404	none
1999	4 Altera 10K100	20	1280	full
1999	Virtex	60	3656	full
2000	Virtex	105	6720	full
1999	Sandia ASIC	145	9280	full
2000	Virtex	168	10752	full

Table 7.11: Performance Ratings of existing DES implementations (source: Patterson [85])

DES applications.

In the ideal situation, only one transport is required to interact with a register in an SLU interface. In the current XC6200 implementation of the FURI core, this is not the case and we must pay a penalty to set up the appropriate device context within the architecture before being able to transport operands to or from SLU interface registers. This is a very relevant constraint to the sequential algorithmic implementation since almost half of the data transports within the soft routing handler's core are handling device context rather than actual operand transports. The first enhancement we suggest is the removal of device context from the host FPGA architecture. By this, we mean that the architecture should no longer require that we make the appropriate map, mask, and wildcard register settings before we can interact with one of the SLU interfaces. Effectively we want to remove state from the logic that controls the configuration memory of the host architecture. If this was done[15], the operand processing cost associated with a single round of DES in the sequential algorithmic model would fall from 48 to 14 instructions (the theoretical maximum in the situation where we do

[15]Chapter 8 discusses and characterises a future device architecture style that would support single transport interactions between SLU interfaces.

not attempt to increase performance through parallelism in the raw transport mechanism). The overall cost for operand processing for the complete algorithm falls, correspondingly, from 768 to 224 instructions[16]. Similarly, removing the data transports associated with device context management in the sea of accelerators and parallel harness implementations reduces the core operand processing costs from 33 and 52 instructions to 23 and 34 instructions respectively.

The high setup costs of the parallel harness and sequential algorithmic models can be tackled by enhancements to the FURI framework. For example, the addition of indexed and indirect addressing support to the FURI core's memory interface SLU will allow us to increase the performance of the configuration protocol. To recall, generating the contents of the code buffer used to transport the actual raw bitstream data from the packet payload to configuration memory is a significant cost incurred by the protocol handler. The best theoretical performance we can hope to achieve for loading configurations was explored in the SLU loading strategies section of Chapter 6. We know from that discussion that precomputing the entire transport sequence required to load the SLU offline gave almost an order of magnitude increase in loader performance. Following that approach for the SLU sizes we have used in this section is generally impractical from the point of view of the spatial resource required to store the elaborated transport sequence and the temporal cost of communicating it. Using indexed and indirect addressing that is explicitly supported by the FURI framework allows us to collapse the instruction costs per configuration word from 96 instructions to a budget of approximately 20 instructions per word, avoiding the use of synthesised code blocks completely. Based on this, we can say that the SLU loading component of the setup cost for each of the models will scale to approximately 20–25% of its earlier cost.

[16] We acknowledge that this requires the underlying architecture to support variable bitwidth transports, but argue this is an acceptable feature to have in a device architecture supporting VC.

Core	Sea of Accel. Original Mbit/sec	Sea of Accel. Enhanced Mbit/sec	Parallel Harness Original Mbit/sec	Parallel Harness Enhanced Mbit/sec	Sequential Alg. Original Mbit/sec	Sequential Alg Enhanced Mbit/sec
19 Cycle, 33MHz	1.6	2.3	1.03	1.57	0.07	0.23
19 Cycle, 100MHz	4.8	6.9	3.09	4.7	0.2	0.69
19 Cycle, 1GHz	48.6	69.7	31.1	47.6	2.05	7.05
8 Cycle, 33MHz	3.8	5.4	2.45	3.73	0.16	0.55
8 Cycle, 100MHz	11.5	16.5	7.4	11.32	0.48	1.6
8 Cycle, 1GHz	92.3	132.4	59.2	90.5	3.9	13.39
2 Cycle, 100MHz	46.17	66.25	29.6	45.3	1.95	6.69
2 Cycle, 400MHz	184.6	264.9	118.3	181	7.81	26.7
2 Cycle, 1GHz	461.7	662.4	295.9	452.6	19.53	66.96
1 Cycle, 1GHz	1108	1590	710.2	1086.3	46.8	160.7

Table 7.12: Projected performance of the VC DES models after device enhancements

Table 7.12 contains the performance projections of the three VC DES implementations, based on our removal of device context and the resulting projected drop in operand processing costs.

7.5 Summary

This chapter has explored the implementation of the three VC models discussed in Chapter 4 on the FURI core. We have seen that there is scope for many different FURI network environments. We then considered how the form of protocol used to communicate with FURI managed programmable logic can be influenced by the communication characteristics of the VC model it supports and the network

environment it is used within. The chapter was then concluded with a discussion of the proposed implementations of the DES in each of the three VC model styles.

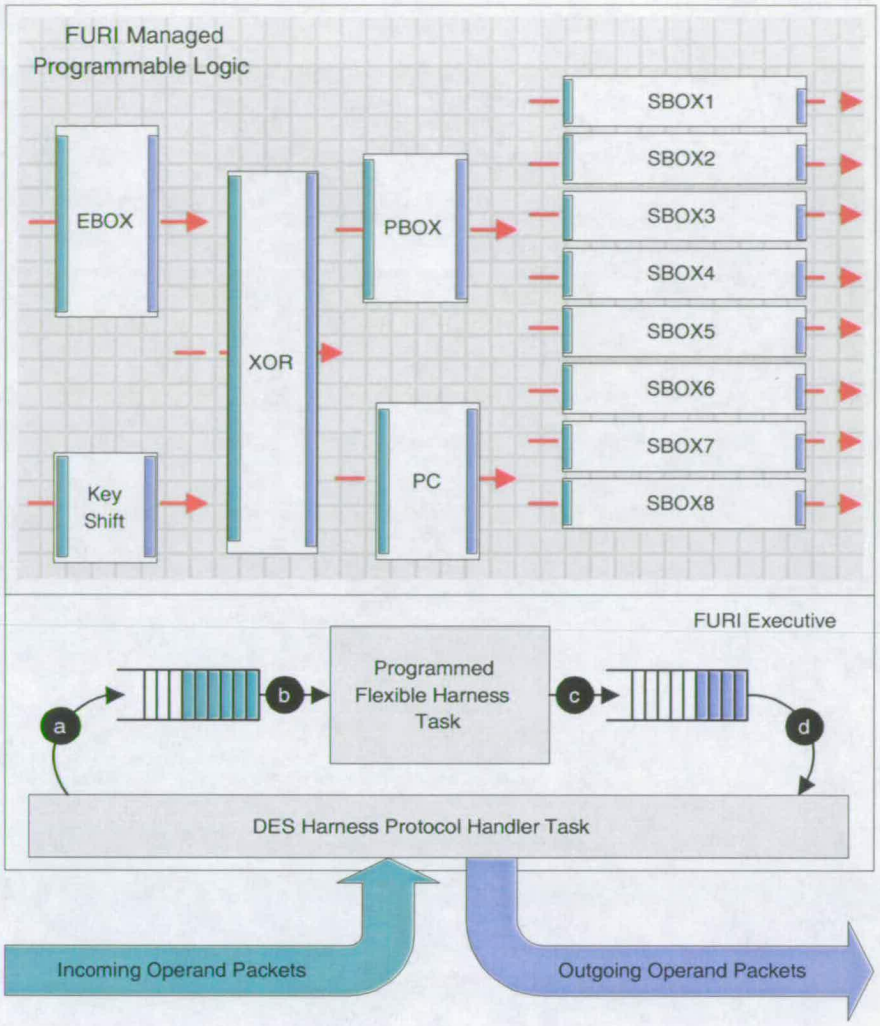

Figure 7.21: Sequential Algorithmic DES: This figure captures the processing stages applied in the FURI environment to support Sequential Algorithmic DES. The FURI executive section holds the two software components of the model. The programmed flexible harness task consumes data packets at stage (b) and produces result packets at stage (c) (the overall packet flow is indicated via the solid black arrows). The DES harness protocol handler task decouples the processing of packet operands from their reception and transmission over external FURI network channels. In stage (a), the task is feeding packets arriving over the FURI network into the flexible harness's processing queue and at stage (d), the task consumes the result packets from the flexible harness task and deals with their transmission. The dashed red arrows are operand transports through each DES SLU, invoked by the flexible harness as it transforms each operand into a result. The programmed execution the flexible harness task ensures each operand flows through the SLUs in the appropriate sequence to implement the DES. For clarity, the diagram does not show the total, connected flow sequence of the operands through every SLU. However, this sequence would be equivalent to a flowchart style abstraction of the Flexible Harness Task's programmed code.

Chapter 8

Conclusions and Further Work

8.1 Overview of Thesis

Chapter 2 introduced the basic features and history of programmable logic devices, and gave particular focus to dynamically reconfigurable FPGAs across two generations of mainstream device architectures.

Chapter 3 discussed the form of mechanisms used to interface and interact with FPGAs within reconfigurable computing systems. At that point we noted how dynamic reconfiguration has encouraged the gradual evolution of the programmable logic device interface in mainstream architectures from their relatively simple, serialised interface origins into more sophisticated parallel interfaces. From there we noted that research device architectures have moved towards application focused, packet oriented, streaming device interfaces. With this backdrop we then considered the concept of a flexible programmable logic device interface capable of adapting to the demands of one of the most compelling uses of dynamic reconfiguration, virtual circuitry.

In Chapter 4, the concept of virtual circuitry was explored in much more detail and we presented the two fundamental VC models. We then described the form and function of an abstract architecture that would be capable of supporting both fundamental VC models, and a third model of our devising.

Chapter 5 presented the design and implementation of an instance of the previous chapter's abstract VC architecture. In particular, the chapter described

the implementation of the Flexible URISC and gave a detailed discussion of its significant design and implementation challenges and their novel solutions.

Chapter 6 looked at the design and runtime environment for the FURI core and paid particular attention to different strategies for loading SLU bitstreams onto the FURI core. The FURI executive was presented as a basic runtime, operating environment and the base protocol used to interact with an operational FURI core was described.

Chapter 7 opened up the discussion of FURI protocols and characterised their form relative to the communication requirements of the different VC models and the influence of different network architectures. The chapter closed with a discussion of the proposed implementations of the DES in each of the three VC styles.

8.2 Contribution

8.2.1 Technical Contribution

The primary technical contribution of this thesis is the novel implementation of the Flexible URISC. In particular, we gave a comprehensive description of the technical requirements, implementation challenges, and corresponding solutions that resulted in the implementation of the first microarchitecture that has an intimate, self-modifying relationship with its host FPGA. Indeed, the technical component in this thesis was first published in [32] and, at the time, was the first detailed paper to tackle the technical challenges, requirements, and approaches to implementing self-modifying circuitry on the only FPGA architecture capable of actually supporting it, the Xilinx XC6200. Furthermore, the technical validity of the approach pioneered in that paper and this thesis has been subsequently reinforced through a small application case study [76] that adopted the same techniques we have described in this thesis.

8.2.2 Conceptual Contribution

Beyond the interesting and novel technical contribution of the Flexible URISC itself, the second, conceptual contribution of this thesis comes from the description of how the unique relationship the FURI core has with its host FPGA architecture can be exploited to implement a flexible, adaptable programmable device interface. In particular, this thesis has described how the implementation of the FURI core and its accompanying runtime environment, the FURI executive, can be used to implement a programmable logic interface that can be adapted to provide support to both of the fundamental models of virtual circuitry plus a third VC model that was previously considered generally impractical.

8.3 Conclusions and Future Directions

8.3.1 Conclusions

There are two broad, immediate conclusions we can draw from the work presented in this thesis. The first is that an implementation of the abstract VC architecture described in Chapter 4 is technically viable, but undertaking such an implementation taxed all of the facilities of the most sophisticated partially reconfigurable FPGA available when this work was carried out in 1996-2000 and still presented many non-trivial technical challenges. The second conclusion we can draw, however, is that overcoming those challenges did produce an implementation of the abstract VC architecture that would support all three of the VC models we described in Chapter 4.

This thesis was written at a very interesting time for runtime reconfiguration research. In 2000, the XC6200 still represents the pinnacle of mainstream, partially reconfigurable device architectures in terms of the facilities it provided to support runtime reconfiguration. As we have mentioned at various points in the thesis, the XC6200 left commercial production in 1998 and its departure essentially marks the end of an era of runtime reconfiguration research. Whilst some

of the subsequent mainstream FPGA architectures are partially reconfigurable to a degree, they provide fewer facilities to support runtime reconfiguration and virtual circuitry than the XC6200. The FPGA architectures of mainstream vendors in 2000 have relatively poor support for partial runtime reconfiguration and lean more towards supporting ASIC replacement and rapid system prototyping.

In the general sense, mainstream vendors and the runtime reconfiguration research community are on divergent paths: device vendors are continually striving to increase the static density of their devices whilst runtime reconfiguration researchers are striving to find the mechanisms that will increase the functional density of their devices through rapid runtime reconfiguration. With the departure of the XC6200, the two most interesting device architectures supporting partial reconfiguration, Colt and Piperench, have come from within the runtime reconfiguration research community itself. The contributions of this thesis are therefore particularly interesting at this point since they essentially constitute a framework for exploring the effectiveness of a new style of programmable logic device interface that is highly adaptive to the demands of different VC models.

The implementation of the FURI core, as presented in this thesis, has relied heavily on the novel features of the XC6200 and an interesting question, therefore, is what effect does the departure of the XC6200 have on potential future implementations of the FURI core? Theoretically, the core could be re-implemented on other mainstream architectures such as the Virtex, provided the system level design could be customised to allow the Virtex access to its own configuration port. The greater challenge in this context would be bridging the gap between the semantics of the Virtex's SelectMap interface and the memory oriented world of the FURI core. The XC6200 implementation has the advantage here because its underlying physical device interface has a natural mapping into the memory oriented world of the FURI core.

To explore the implementation of FURI on the Virtex architecture further we

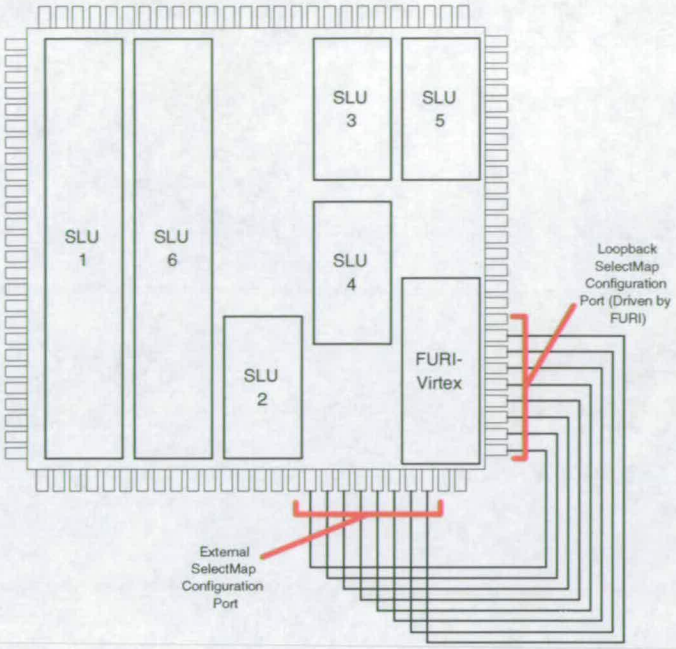

Figure 8.1: FURI Virtex: remapping the configuration port

must overcome at least the two challenges identified above. The first of these, taking control of the device's configuration port, requires quite a different approach to that taken for the XC6200 implementation. The XC6200 supports access to its configuration port from within the device itself but the Virtex architecture does not[1]. However, this does not mean that all is lost. Rather, we must resort to physically re-mapping the Virtex's own control port to user pins that the FURI core within the device could use subsequently to drive it's host configuration interface. This organisation is shown in Figure 8.1. Once we have achieved this, however, the device pins driven by the FURI core respond in a comparable manner to the `cbuf` components that we instantiated for the XC6200 FURI implementation.

Another important consideration builds on the column-based reconfigurability of the Virtex architecture. In the XC6200, function units can be reconfigured in isolation, independent of any other parts of the device. However, since we can only reconfigure on a column by column basis in the Virtex, we are encouraged

[1] An internally accessable configuration port is a listed feature of the Virtex II architecture, although this version of the architecture is not yet available in 2000.

to make the geometric area occupied by the Virtex FURI core as lean as possible. Increasing the spread of the core's circuitry over successive columns means we must take greater care when reconfiguring those columns to ensure that the core circuitry is not adversely affected by the configuration. By this rationale, we should consider the placement of SLU 5 in Figure 8.1 as dangerous: we must take care to preserve both the infrastructure (routing and CLB allocations) and the context within any state elements of the column. Overall, this problem is similar to the problems we discussed in Chapter 6 on overlaying SLUs with the XC6200 FURI's core circuitry. Here, however, we can see that the problem exists at a much coarser granularity.

Just as we used the FastMap interface of the XC6200 to reconfigure the device and interact with SLUs, we would also use the Virtex's SelectMap interface to access SLU inputs and outputs. However, we must acknowledge that this task is significantly more complex in the Virtex architecture. Rather than reading just the bits of the register we require, we must read the entire configuration sequence of a column and then extract the pertinent bits that reveal the register state. Essentially, much of the processing that was done implicitly within the XC6200 architecture must now be done explicitly by extensions to the FURI core's memory interface. In particular, the layer of logic that we place between the FURI core's memory buses and the SLUs must grow to allow us to transport operands to and from the SLUs.

Besides this complication, however, there are some interesting advantages to a FURI-Virtex implementation. Firstly, the Virtex architecture supports much faster logic circuit implementations, far in excess of the clock speeds that can be achieved on a XC6200 implementation. From this, FURI cores operating in the scale of hundreds of megahertz are very achievable. Furthermore, a FURI Virtex implementation can take advantage of architectural features that facilitate interaction between circuits operating at different clock speeds. For example, the

architecture supports compact implementations of multi-ported memories that operate in FIFO mode. As we alluded to in our discussions of the DES implementation in Chapter 7, these embedded memory blocks are very effective for bridging timing differences between clock domains. Other wide embedded memories in the architecture allow us to support internal code regions within the architecture without explicitly consuming cell resources. This is relevant since on-chip memory blocks are much more effective at supporting instruction streams for the high speed FURI cores we would expect to implement.

Although there are issues regarding the FURI Virtex's ability to interact with SLUs, we cannot advocate the XC6200's memory interface implementation as the ideal. One of the XC6200 implementation of the FURI core's strengths is to interleave SLU configuration, communication and computation on the granularity of single data transfers. This challenges a fundamental premise in many implementations of VC – that relatively long periods of time must be dedicated exclusively to SLU configuration. This has encouraged the view of the whole bitstream as the granular unit of configuration. Additionally, the commonality of closed, proprietary bitstream formats, encourages the designer of reconfigurable systems to abstract away from the low level aspects of SLU loading and ignore the fact that loading each individual datum from an SLU bitstream is a context sensitive process.

We witnessed this, in the case of the XC6200, through the bitstream's dependence on the correct map, mask, and wildcard register settings. The discussion on SLU loading in Chapter 6 described how these features of the XC6200's FastMap interface makes the FURI core's task more complex. Although the FURI core is capable of interleaving data transfers at such fine granularities we cannot treat the configuration bitstream as a sequence of individual, context-free data transfers. The compromise between the two granularities, also discussed in Chapter 6, is therefore to define the unit of configuration according to the structure within the

SLU bitstream itself. One observation, however, is that the compelling technique of configuration compression [49] increases the influence of wildcarding within a bitstream and, as a consequence, actually creates more context dependencies between individual data transfers and the device state.

A prospective solution for future FPGA architectures that we would like to suggest in this thesis involves moving away from interacting with SLUs based on their geometric location within the array and towards support for a symbolic mapping[2] to SLU ports. Such a "SymbolMap" interface would essentially bind a symbolic reference to each input and output register of an SLU as it is instantiated onto the array fabric. Any future references to the SLU interface are done by reading and writing to symbolic names of the target SLU's interface registers. We acknowledge this approach would involve paying a physical cost to add the functionality to the array resource. Prospectively, we would be making parts of the configuration memory of the host FPGA respond as slightly complex content addressable memories. However, the elimination of a significant proportion of device control state handling from the FURI framework leaves it free to orchestrate the flow of operands and configurations over the reconfigurable resource.

8.3.1.1 Supporting Variant Virtual Circuitry Models

Besides the three main VC models considered in this thesis, we can identify other notable VC models, such as the Virtual Pipeline [69] in particular, that are interesting to relate to the FURI framework. Virtual Pipelines are strategies applied to the reconfiguration of regular pipelined and systolic style virtual circuits. The aim of the strategies are to minimise the latency incurred when reconfiguring the pipeline. This is achieved by overlapping the configuration of each pipelines stage within ongoing computation within the unaltered stages. Rather than flushing the entire pipeline of its data, reconfiguring it in its entirety, and re-filling it with new operand data, the pipeline gradually "morphs" between full configura-

[2] By symbolic, we mean a logical reference encoded as an integer.

tions. Data from the original pipeline circuitry continues to flow through what remains of that circuitry, ahead of the point of reconfiguration. Behind the point of reconfiguration, the new pipeline structure has been established and begins to immediately receive new operand data.

Fundamentally, this model is a close relative to the parallel harness VC model we have discussed but with a degree of reconfigurability targeted at the SLUs within the wired harness. FURI support for the parallel harness model was discussed Chapter 7. We can generalise that discussion and consider how we could take advantage of the reconfigurability provided by FURI to support the virtual pipelines. In [69], the Luk and Shirazi state that the cost of reconfiguring a pipeline stage within the virtual pipeline should ideally be balanced with the processing rate of the pipeline as a whole. The FURI core is in an ideal position to effect this style of reconfiguration: its close coupling to the configuration port of the virtual pipeline's host FPGA means that we can effect the reconfigurations without incurring physical latencies in the system architecture. In FURI terms, the act of morphing a single pipeline stage is equivalent to loading a new SLU over an existing SLU in the parallel harness. As we mentioned previously in the thesis, the three VC models that we gave particular emphasis too should not be taken as the definitive set of models: the virtual pipeline is an example of the fluidity of definition that exists between the different models as we trade off the degrees of reconfigurability and degrees of SLU interaction.

From the discussion above, we can see how additional conceptual VC models map to the FURI framework. Also, the discussion on the FURI Virtex implementation, demonstrates how the framework itself can map onto newer generations of device architecture. Besides this, it is also interesting to address the relationship between FURI and other platforms for virtual circuit models. The Piperench architecture we described in Chapter2 is a particularly relevant here: essentially, it is a hardware implementation of a one dimensional virtual pipeline. We saw in

Chapter 2 how Piperench's architecture is specifically designed for SLUs in the form of pipeline stages. Further, the architecture's reconfiguration facilities explicitly support the reconfiguration of pipeline stages in the incremental manner required by the virtual pipeline model: the device architecture can rapidly move the configuration data used to implement one stage to a different part of the reconfigurable fabric. In terms of raw performance, the Piperench architecture is likely to exceed that of a basic FURI implementation of a virtual pipeline. Since the performance difference is the result of tailoring the physical architecture of the FPGA platform, this effectively constrains the device to the one virtual circuitry model. FURI is unlikely to compete in terms of raw performance for the main enumerations of the virtual pipeline model that Piperench targets, but its overall performance for a series of VC models will be higher. Furthermore, even within the virtual pipeline model, there are potentially malignant cases that will map better to a FURI implementation because they, for example, do not map well to the reconfigurable stripes of Piperench.

8.3.1.2 Meeting the requirements for Virtual Circuits

In Chapter 4 we outlined some basic requirements for supporting virtual circuits. We now consider how well these were met by the FURI framework. There are three points that we will make regarding this. The first here is that the XC6264 implementation of the FURI core produced for this thesis meets the 10% resource utilisation constraint.

Further to this, we can consider how well we can configure the host FPGA using the FURI framework with respect to the earlier constraint on saturating the host FPGA's configuration port. The basic, 33MHz FURI implementation has a transport cycle time of 2.6μs which is above our target rate of 40ns. The requirement is met for a single-cycle 33MHz implementation of the core whose corresponding transport cycle time is 30ns. Successive increases in clock speed reduce the degree of pipelining required. For example, the 200MHz, 8 cycle core

has a transport cycle time of exactly 40ns.

Finally, we can note that driving SLU interfaces at the core circuitry speed is a much more demanding requirement, that we cannot meet with the current generation of device architectures. The interfaces that FPGAs provide to interact with SLUs typically operate at much lower speeds than the circuitry itself. The XC6200 is more benevolent than the Virtex architecture in this matter, but it still lags behind the bandwidths that would be required to access, for example, a 66MHz pipelined DES SLU.

8.3.2 Future Directions

Chapter 7 proposed FURI implementations of the DES in the style of each of the VC models with the aim of demonstrating that the FURI core can indeed support all three VC models. The first extension to this work would be a full implementation of each of the VC models with the aim of quantifying the relative effectiveness of different FURI protocols when they are used for particular applications and communication traffic patterns.

The aim of this thesis was not to present a particularly high performance implementation of the FURI core, its kernel circuitry, or its surrounding system. However, the performance of the FURI core circuitry could be increased through pipelining, but this must be carefully balanced against the number of memory ports available within the system. There is also room to explore alternative scheduling mechanisms within the FURI executive using supporting circuitry to aid the decision process. At various points in the thesis we described potential optimisation techniques that would be applied by a higher level FURI compiler. Implementing such a compiler was beyond the scope of this thesis, but the FURI core's membership of the transport triggered architecture class means future work in this area has the potential to exploit existing TTA compiler technology [54].

The FURI core has the potential to become a testbed for an emerging design

methodology for runtime reconfiguration that is based on the notable philosophy presented in [15]. Instead of attempting to extend traditional hardware-software co-design into the reconfigurable domain, this paper advocates a control flow-data flow co-design methodology where the system design is repeatedly partitioned into control and dataflow components. Here, rather than making a single, initial partition of the system into hardware and software components. In the control-dataflow methodology described here, we allow components to be re-partitioned into sub-components elements of control-flow or data-flow with each level of system decomposition. This is different from the predominantly static system partitioning undertaken once at the beginning of the hardware-software co-design process. Furthermore, it creates a close interplay between control flow components and data flow components within the system hierarchy and requires an efficient and effective control flow-data flow interface. The FURI system described in this thesis is relevant to this methodology because it comprises elements of control flow, elements of data flow, and approaches the issues related to maintaining an effective and tightly integrated control flow-data flow interface. An historical architecture, the ICL2900 with the Distributed Array Processor(DAP)[86], is also notable here for its combination of tightly coupled processor and programmable array core.

8.4 Conclusion

When Kean introduced the first partially reconfigurable FPGA in 1989, he concluded his thesis with the following statement:

> Configurable logic was an idea that arrived before its time: now that its time has come it would be a pity to go on ignoring it.

In the decade between the introduction of the CAL architecture and the presentation of this thesis, configurable logic has been anything but ignored: in

2000, configurable logic is a multi-million dollar industry. FPGAs have become a well established technology for ASIC replacement and rapid system prototyping, but the most compelling use of partially reconfigurable FPGAs, runtime reconfiguration, has remained the most elusive. The challenge in 2000 is to take the experiences of the past decade and define the new form device architecture and, particularly device interface, that will transport runtime reconfiguration from being a delicate, niche technique into mainstream acceptance.

Appendix A

FURI Core Implementation Details

A.1 Introduction

This appendix describes the status of the implementations of the components of the FURI framework discussed in the main body of the thesis.

A.2 The FURI Core

The main core of the Flexible URISC, described in Chapter 5, has been fully implemented on both the XC6216 and XC6264 versions of the XC6200. The final implementation produced through this thesis used 860 XC6200 function units which is approximately 20% of the available function units in a host XC6216 and 5% in a host XC6264. Figure A.1 shows the actual layout of the implemented FURI core on a XC6264. Although the majority of the cells are not consumed in the XC6216 implementation, it is difficult to utilise the unoccupied cells for dynamically instantiating SLUs. The routing resources of the XC6216 are heavilly utilised by the FURI SRAM address and data buses that traverse the array. Any SLU that would be instantiated alongside the FURI core would have to be carefully overlayed with the existing core routes. However, the XC6200's routing resources become congested as the FURI SRAM signals approach the device IOBs. Since these signals begin to consume the same low level routing resources that

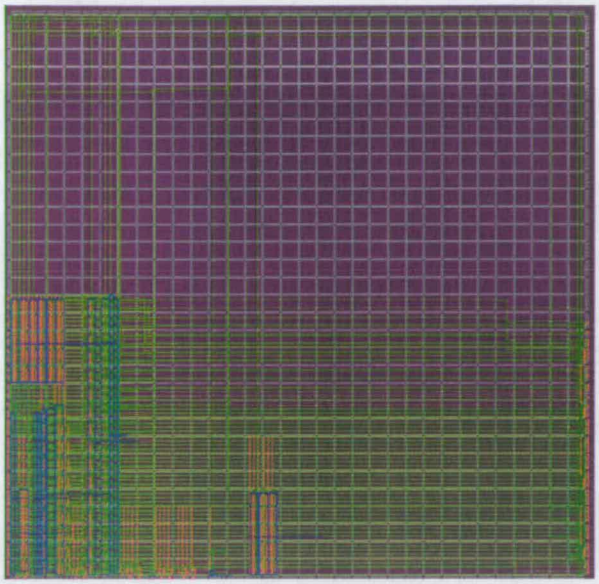

Figure A.1: Placed and Routed Layout of the FURI core on a Xilinx XC6264

are predominantly targeted by FURI SLUs, it is difficult to dynamically overlay the SLUs without affecting the underlying core cirtuitry.

A.3 The FURI Assembler

The FURI assembler is the central tool in the FURI design flow that we introduced in Chapter 7. In this section we will give an expanded description of its operation. As mentioned earlier, the FURI assembler is "macro" based. It accepts a stream of instructions and instruction definitions (instruction macros) and translates the instruction stream into a FURI executable.

A.3.1 Basic Assembly Constructs

The fundamental constructs the FURI assembler accepts in its input stream are macro definitions, code blocks, data literal definitions, and assembler pragmas.

A.3.1.1 Macros

The assembler understands that the single fundamental instruction in the code stream is the single-word `move`. This forms the root of a tree of instruction defini-

```
.begin macro <macro_name> ( <operand names> )

move <addr>, <addr>
instrA lit1, op2, lit2
.label1 instrB op1
instrC op1, label1, lit3, lit4
.end macro
```

Figure A.2: The basic format of a macro definition

tions and, as macros are introduced, the set of available instructions expands. A macro definition takes the form given in Figure A.2. The first line of the definition identifies the name of the instruction and the exact series of operators that are required to satisfy the instantiation of that macro. The code section of the macro can contain a mixture of basic move instructions and other higher level macro instructions. In this way, more complex hierarchical instruction definitions can be created from lower level sequences of instructions. The only restriction on this hierarchical composition is that macro instructions referenced within the body of the macro must have been declared earlier.

Instructions within the body of the macro can make reference to any globally defined data literals, labels within the body of the macro itself, and any of the macro's parameter operands. In addition to these, a special symbol "next" is provided. When this symbol is referenced within the body of the macro it is replaced with the address of the instruction that will follow the instantiated macro. This functionality is required specifically to support branching instructions where it is necessary to jump either to the branch address or to the address of the instruction following the branch instruction itself. Labels on macro instructions are treated specially during the code generation phase. Each time an instruction macro is instantiated, versions of the instruction labels specific to that instantiation are also synthesised and all internal references to the label are redirected to the synthesised label.

A.3.1.2 Code Blocks

Code blocks are instruction sequences that will eventually be instantiated into the memory image. Figure A.3 contains a code block from the source tree of the FURI executive. Using the assembly pragmas discussed below, the assembler is aware of which address in the memory image that a code block should be placed at. The amount of space required per instruction can be calculated by referring to the macro that will be used to implement that instruction. Instructions within a code block can be prefixed with labels which are entered into a global symbol table and can be referred to as operands in other instructions.

A.3.1.3 Data Literals

Data literals are supported by the assembler in two ways. Firstly, sections of the memory image can be explicitly reserved for a literal using the `.literal` directive. This places an entry for the literal in the symbol table and can also be used to specify a particular location in the memory image for the literal to be placed. In addition to this, the directive can also identify a default value for the literal.

The second route for introducing data literals is through implicit references within the operand lists of an instruction. In Chapter 5, we discussed how the FURI architecture has no immediate addressing mode. The FURI assembler, as implemented for this thesis, supports immediate addressing of operands within the assembly source and, for each immediate operand, synthesises a new entry in the symbol table, reserves a location in the memory image, and ultimately populates that location with the correct (immediately addressed) data value. A C-language style "address of" operator is implemented in a similar manner.

A.3.1.4 Assembly pragmas

Embedded within the source files parsed by the FURI assembler are a series of directives. These pragmas influence different stages of the assembly process and

```
.begin code add_task

;; Which address can I put this task at?
.add_task add #0x1, freeNodePtr, fnPtrAddr

move_da fnPtrAddr, freeNodeAddr
add #0x1, freeNodeAddr, freeNodePRVAddr
add #0x2, freeNodeAddr, freeNodeNXTAddr

;; ASSERT the last node added is pointed to by the global
;; lastTaskInserted
move lastTaskInserted, prevNodeAddr
add #0x1, prevNodeAddr, prevNodePRVAddr
add #0x2, prevNodeAddr, prevNodeNXTAddr

;; Store the exe addr in the EXE field of the free node
move_db newTaskAddr, freeNodeAddr

;; Store the addr of the prevNode in the PRV field...
move_db prevNodeAddr, freeNodePRVAddr

;; Store the NXT field from the prevNode as the NXT field
;; in this node
move_dab prevNodeNXTAddr, freeNodeNXTAddr

;; Modify the fields of the prevNode NXT field so it now
;; points to the new node...
move_db freeNodeAddr, prevNodeNXTAddr

;; I've consumed a cell in the freeTaskNodes, reduce the
;; stackpointer to compensate...
move_da freeNodePtr, freeNodePtr

;; point to the new node as the last one to be inserted
move freeNodeAddr, lastTaskInserted

.end_add_task ret 0x0
.end
```

Figure A.3: A FURI Assembler Code Block

generally provide information about the size of the FURI memory image and the locations at which code blocks and data segments should be placed. Evocations of the pragmas in the assembly source are prefixed by the `.pragma` directive followed by the name of the particular pragma being used. The effect of the three supported pragmas is described below.

map_device This pragma is used to describe which address regions of the potential 32-bit address space are actually populated. During the code generation phase, the assembler checks that each population of the memory image is to a region that has actually been mapped to a memory device.

data_segment The data_segment pragma identifies a point in the memory image where literals will be placed at if they do not have an explicit address associated with them.

load_point The load_point pragma has a similar effect to the data_segment pragma but influences where the elaborated code blocks will be placed in the memory image.

A.3.2 Outline of the Assembly Process

There are three phases to the assembly process. In the first phase, the assembler parses all of the source files and their dependancies. The result of this is the creation of two internal datastructures: a macro list to hold the set of currently available instruction definitions; and a code fragment list to track information on which instructions and data literals are to be instantiated into the FURI memory map of the output executable. During this initial parsing phase, there are no strict requirements on the declaration of symbolic labels before they are referenced. Forward references to labels and symbols within the assembly code is a basic functionality that is supported.

The second phase of the assembly process is code generation. Here, each of the instructions are elaborated down to a set of fundamental move instructions

and placed within an internal representation of the FURI memory image. Data literal definitions are also inserted into the memory image at this stage. To elaborate an instruction down to it's sequence of data transport moves involves locating the correct instruction macro definition and then creating a new instance of the macro's primitive instruction template. This is specialised to the particular destination address that the instantiated instruction will have in memory and the addresses of the data literals that were supplied as operands to the higher level instruction. Before an instruction is presented for elaboration, it's symbolic references (if there were any that could not be resolved when the instruction was first parsed) are resolved. This allows all of the references to operands within the macro's code template to be matched to the actual addresses of the data literals in memory. Once a specialised instance of the macro's code template has been created, it's instruction sequence (which is now a fully elaborated set of primitive move instructions with no symbolic references remaining) is written to the appropriate segments of the FURI memory image.

In the third assembly phase the memory image produced through code generation is dumped out in one of three main formats: a core dump of the entire memory region; a command file for `qPCItest`; or a data stream file for use in FURI protocols. Side products of this phase include symbolic debugging data such as a symbol table dump and code footprint dump. Both of these files can be used within `qPCItest` for interrogating the actual, executing memory image of the FURI core.

A.3.3 Assembling FURI Protocol code

To assemble FURI protocol code, the assembler supports two main funcionalities: the linking of symbolic information from previously-assembled code to the code currently being assembled; and the conversion and output of the final, assembled memory image as a stream of data words that can be transmitted as packets over

the FURI base protocol.

Linking to previously assembled code is necessary here as the code transmitted over the FURI base protocol will require access to the services of, for example, the FURI executive. Since these involve subroutine calls and access to pre-defined literals, the protocol code is given access to the symbol table of the FURI executive through the assimilation of the symbolic debugging information described above into the symbol table when the protocol code is transmitted.

The generation of a dataword stream is different from the approach taken by the qPCItest command script generator. Commands scripts make more assumptions about the physical, board RAM having a known state initially. Because of the dynamic nature of the code's transmission, a datastream does not make any assumptions about the board's RAM state.

Bibliography

[1] *Ieee international workshop on rapid system prototyping (rsp)*, IEEE Computer Society.

[2] *Ieee symposium on field programmable custom computing machines (fccm)*, IEEE Computer Society.

[3] *International workshop on field-programmable logic and applications (fpl)*, Springer.

[4] *MPACF250 - MAP's CORE+ Reconfigurable System*, Product Brief, April 1998.

[5] Algotronix, Ltd., *Cal1024 datasheet*, 1990.

[6] R. Amerson, R. Carter, B. Culbertson, P. Kuekes, and G. Snider, *Teramac–configurable custom computing*, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (Napa, CA) (D. A. Buell and K. L. Pocek, eds.), April 1995, pp. 32–38.

[7] J.M. Arnold, D.A. Beull, and E.G. Davis, *Splash 2*, Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures, June 1992, pp. 316–324.

[8] Atmel, Configurable Logic Data Book, Atmel Corporation, San Jose, CA, 1997.

[9] Atmel Corporation, *At40k FPGAs with FreeRAM^{tm}*, 1999, Datasheet.

[10] P. Bellows and B. L. Hutchings, *JHDL - an HDL for reconfigurable systems*, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (Napa, CA) (J. M. Arnold and K. L. Pocek, eds.), April 1998, pp. 175–184.

[11] R. Bittner and P. Athanas, *Wormhole run-time reconfiguration*, ACM/SIGDA International Symposium on Field Programmable Gate Arrays (Monterey, CA), February 1997, pp. 79–85.

[12] G. Brebner, *A Virtual Hardware Operating System for the Xilinx XC6200*, Proc. 6th International Workshop on Field-Programmable Logic and Applications, FPL'96 (Darmstadt, Germany) (R. W. Hartenstein and Manfred Glesner, eds.), Springer-Verlag, September 1996, pp. 327–336.

[13] ———, *Automatic identification of swappable logic units in XC6200 circuitry*, Field-Programmable Logic: Smart Applications, New Paradigms and Compilers. 7th International Workshop on Field-Programmable Logic and Applications, FPL '97 (London, United Kingdom) (W. Luk, P. Cheung, and R. W. Hartenstein, eds.), Springer-Verlag, September 1997, pp. 173–182.

[14] ———, *The Swappable Logic Unit: a Paradigm for Virtual Hardware*, IEEE Symposium on FPGAs for Custom Computing Machines (K. L. Pocek and J. M. Arnold, eds.), IEEE Press, April 1997.

[15] ———, *Field-programmable Logic: Catalyst for New Computing Paradigms*, Field Programmamble Logic and Applications – From FPGAs to Computing Paradigm (A. Keevallik R.W. Hartenstein, ed.), vol. 1482, 1998, pp. 49–48.

[16] G. Brebner and A. Donlin, *Runtime Reconfigurable Routing*, Parallel and

Distributed Processing (José Rolim, ed.), LNCS, vol. 1388, Springer-Verlag, 1998, pp. 25–30.

[17] G. Brebner and J. Gray, *Use of reconfigurability in variable-length code detection at video rates*, Field-Programmable Logic and Applications. 5th International Workshop on Field-Programmable Logic and Applications (Oxford, UK) (W. Moore and W. Luk, eds.), Springer-Verlag, September 1995, pp. 429–438.

[18] D. Burger and J. R. Goodman, *Billion-transistor architectures*, IEEE Computer, vol. 30, IEEE Computer Society, Sep 1997, pp. 46–50.

[19] J. Burns, A. Donlin, J. Hogg, S. Singh, and M de Wit, *A dynamic reconfiguration run-time system*, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (Napa, CA) (J. Arnold and K. L. Pocek, eds.), April 1997.

[20] S. Cadambi, J. Weener, S. C. Goldstein, H. Schmit, and D. E. Thomas, *Managing pipeline-reconfigurable FPGAs*, ACM/SIGDA International Symposium on Field Programmable Gate Arrays (Monterey, CA), February 1998, pp. 55–64.

[21] S. Churcher, T. Kean, and B. Wilkie, *The XC6200 FastMap^{TM} processor interface*, Field-Programmable Logic and Applications : 5th International Workshop (Oxford, United Kingdom) (W. Moore and W. Luk, eds.), LNCS, vol. 975, Springer-Verlag, August/September 1995, pp. 36–43.

[22] D. A. Clark and B. L. Hutchings, *Supporting FPGA microprocessors through retargetable software tools*, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (Napa, CA) (J. Arnold and K. L. Pocek, eds.), April 1996, pp. 195–203.

[23] H. Corporaal, *A different approach to high performance computing*, Proceedings of the International Conference on High Performance Computing, 1997.

[24] ———, *Microprocessor architectures from vliw to tta*, John Wiley, 1998.

[25] Actel Corporation, *Introduction to the Actel FPGA architecture*, Application Note, April 1996.

[26] ———, *ProASIC^{tm} 500K Family*, Product Datasheet, December 1999.

[27] Altera Corporation, *Altera FLEX10K embedded programmable logic family*, June 1996, Product Datasheet, Version 2.

[28] Xilinx Corporation, *XAPP015: Using the XC4000 Readback Capability*, Application Note.

[29] ———, *XAPP122: The Express Configuration of SpartanXL FPGAs*, Application Note, November 1998.

[30] Xilinx Development Corporation, *Xc6200 development system*, January 1998, Datasheet, Version 1.2.

[31] A. DeHon, *DPGA-coupled microprocessors: Commodity ICs for the early 21st century*, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (Napa, CA) (D. A. Buell and K. L. Pocek, eds.), April 1994, pp. 31–39.

[32] A. Donlin, *Self Modifying Circuitry - A Platform for Tractable Virtual Circuitry*, Field Programmamble Logic and Applications – From FPGAs to Computing Paradigm, 8th International Workshop, FPL'98 (A. Keevallik R.W. Hartenstein, ed.), vol. 1482, Springer-Verlag, 1998, pp. 199–208.

[33] C. Ebeling, G. Borriello, S. A. Hauck, D. Song, and E. A. Walkup, *TRIPTYCH: a new FPGA architecture*, FPGAs. International Workshop on Field Programmable Logic and Applications, September 1991, pp. 75–90.

[34] H. Eggers, P. Lysaght, H. Dick, and G. McGregor, *Fast reconfigurable crossbar switching in FPGAs*, Field Programmable Logic: Smart Applications, New Paradigms and compilers, 6th International Workshop, FPL'96 (R. W. Hartenstein and M. Glesner, eds.), LNCS, vol. 1142, Springer-Verlag, 1996, pp. 297–306.

[35] G. Estrin, *Organization of computer systems – the fixed plus variable structure computer*, Proceedings of the Western Joint Computer Conference, 1960, pp. 33–40.

[36] G. Estrin, B. Bussell, R. Turn, and J. Bibb, *Parallel processing in a restructurable computer system*, IEEE Transactions on Electronic Computers (1963), 747–755.

[37] A. Marshall et al., *A reconfigurable arithmetic array for multimedia applications*, FPGA'99, ACM/SIGDA International Symposium on Field Programmable Gate Arrays, ACM Press, February 1999, pp. 135–143.

[38] E. Waingold *et al*, *Baring it all to Software: Raw Machines*, IEEE Computer **30** (1997), no. 9, 86–93.

[39] J. Faura, J. M. Moreno, M. A. Aguirre, P. van Doung, and J. M. Insenser, *Multicontext dynamic reconfiguration and real-time probing on a novel mixed-signal programmable device with on-chip microprocessor*, Field-Programmable Logic and Applications : 7th International Workshop, FPL'97 (W. Luk, P. Y. K. Cheung, and M. Glesner, eds.), LNCS, vol. 1304, Springer-Verlag, 1997.

[40] P. W. Foulk, *Data-folding in SRAM configurable FPGAs*, IEEE Workshop on FPGAs for Custom Computing Machines (Napa, CA) (Duncan A. Buell and Kenneth L. Pocek, eds.), IEEE Computer Society Press, April 1993, pp. 163–171.

[41] P. C. French and R. W. Taylor, *A self-reconfiguring processor*, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (Napa, CA) (D. A. Buell and K. L. Pocek, eds.), April 1993, pp. 50–59.

[42] M. Gokhale, W. Holmes, A. Kosper, D. Kunze, D. Lopresti, S. Lucas, R. Minnich, and P. Olsen, *SPLASH: A reconfigurable linear logic array*, International Conference on Parallel Processing, 1990, pp. I–526–I–532.

[43] G. W. Griffin, *The Ultimate Ultimate RISC*, Computer Architecture News (1988).

[44] S. A. Guccione and D. Levi, *XBI: A java-based interface to FPGA hardware*, Configurable Computing: Technology and Applications, Proc. SPIE 3526 (Bellingham, WA) (John Schewel, ed.), SPIE – The International Society for Optical Engineering, November 1998, pp. 97–102.

[45] ———, *Run-time parameterizable cores*, Field-Programmable Logic and Applications, 9th International Workshop, FPL'99 (Patrick Lysaght, James Irvine, and Reiner W. Hartenstein, eds.), LNCS, vol. 1673, Springer-Verlag, Berlin, August/September 1999, pp. 215–222.

[46] J. D. Hadley and B. L. Hutchings, *Design methodologies for partially reconfigured systems*, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (Napa, CA) (P. Athanas and K. L. Pocek, eds.), April 1995, pp. 78–84.

[47] I. Hadžić, S. Udani, and J. M. Smith, *FPGA viruses*, Field-Programmable Logic and Applications, 9th International Workshop, FPL'99 (Patrick

Lysaght, James Irvine, and Reiner W. Hartenstein, eds.), LNCS, vol. 1673, Springer-Verlag, Berlin, August/September 1999, pp. 291–300.

[48] S. Hauck, *Configuration Prefetch for Single Context Reconfigurable Coprocessors*, FPGA'98, ACM/SIGDA International Symposium on Field Programmable Gate Arrays, ACM Press, 1998, pp. 65–74.

[49] S. Hauck, X. Li, and E. Schwabe, *Configuration compression for the Xilinx XC6200 FPGA*, IEEE Symposium on Field-programmable Custom Computing Machines (J. M. Arnold and K. L. Pocek, eds.), IEEE Press, 1998.

[50] S. Hauck and W. D. Wilson, *Runlength compression techniques for FPGA configurations*, IEEE Symposium on FPGAs for Custom Computing Machines (Napa, CA) (Kenneth L. Pocek and Jeffrey Arnold, eds.), IEEE Computer Society Press, April 1999, pp. 286–287.

[51] D. Hawley, *Advanced PLD architectures*, FPGAs (Oxford, England) (W. Moore and W. Luk, eds.), Abingdon EE and CS Books, September 1991, pp. 11–23.

[52] J. Hogg, *A Dynamic Hardware Generation Mechanism based on Partial Evaluation*, Designing Correct Circuits (M. Sheeran and S. Singh, eds.), Springer Electronic Workshops in Computing, 1996.

[53] S. H. Hollingdale, *High speed computing; methods and applications*, English University Press, 1959.

[54] J. Hoogerbrugge and H. Corporaal, *Code generation for transport triggered architectures*, Code Generation for Embedded Processors (Gert Goossens and Peter Marwedel, eds.), 1995, pp. 240–259.

[55] P. James-Roxby and E. Cerro-Prada, *A wildcarding mechanism for acceleration of partial configurations*, Field-Programmable Logic and Applica-

tions, 9th International Workshop, FPL'99 (P. Lysaght, J. Irvine, and R. W. Hartenstein, eds.), LNCS, vol. 1673, Springer-Verlag, Berlin, August/September 1999, pp. 444–449.

[56] P. James-Roxby, E. Cerro-Prada, and S. Charlwood, *A core-based design method for reconfigurable computing applications*, IEE Informatics Colloquium on Reconfigurable Systems, Institute of Electrical Engineers, March 1999, pp. 3/1 – 3/4.

[57] D. W. Jones, *The Ultimate RISC*, Computer Architecture News **16** (1988), no. 3, 48–55.

[58] T. Kean, *Configurable logic: A dynamically programmable cellular architecture and its VLSI implementation*, Ph.D. thesis, Department of Computer Science, University of Edinburgh, 1989.

[59] T. Kean and A. Duncan, *A 800Mpixel/sec reconfigurable correlator on the XC6216*, Field-Programmable Logic: Smart Applications, New Paradigms and Compilers, 7th International Workshop, FPL '97 (London, United Kingdom) (W. Luk, P. Cheung, and R. W. Hartenstein, eds.), Springer-Verlag, September 1997, pp. 382–391.

[60] R. Kress, *A fast reconfigurable ALU for Xputers*, Ph.D. thesis, Kaiserslautern University, 1996.

[61] R. Laufer, R. Reed Taylor, and H. Schmit, *PCI-PipeRench and the SwordAPI: A system for stream-based reconfigurable computing*, IEEE Symposium on FPGAs for Custom Computing Machines (Napa, CA) (Kenneth L. Pocek and Jeffrey Arnold, eds.), IEEE Computer Society Press, April 1999, pp. 200–208.

[62] S. H. Lavington, *Manchester Mark I and Atlas: A historical perspective*, Communications of the ACM **21** (1978), no. 1, 4–12.

[63] M. Leeser, W.M. Meleis, M.M. Vai, and P. Zavracky, *Rothko: a three-dimensional FPGA architecture, its fabrication, and design tools*, Field-Programmable Logic: Smart Applications, New Paradigms and Compilers. 7th International Workshop on Field-Programmable Logic and Applications, FPL '97 (London, United Kingdom) (W. Luk, P. Cheung, and R. W. Hartenstein, eds.), Springer-Verlag, September 1997, pp. 21–30.

[64] D. Levi and S. A. Guccione, *BoardScope: A debug tool for reconfigurable systems*, Configurable Computing: Technology and Applications, Proc. SPIE 3526 (Bellingham, WA) (John Schewel, ed.), SPIE – The International Society for Optical Engineering, November 1998, pp. 239–246.

[65] Z. Li and S. Hauck, *Don't care discovery for FPGA configuration compression*, ACM/SIGDA International Symposium on Field Programmable Gate Arrays (Monterey, CA) (S. Kaptanoglu and S. Trimberger, eds.), ACM SIGDA, ACM Press, February 1999, pp. 91–98.

[66] A. Lovelace, *Sketch of the Analytical Engine invented by Charles Babbage, by L. F. Menebrea of Turin, with notes on the memoir by the translator*, Taylor's Scientific Memoirs **3** (1843), no. 29, 666 – 731.

[67] W. Luk, N. Shirazi, and P. Y.K. Cheung, *Modelling and optimising run-time reconfigurable systems*, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (Napa, CA) (J. Arnold and K. L. Pocek, eds.), April 1996, pp. 167–176.

[68] ———, *Compilation tools for run-time reconfigurable designs*, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (Napa, CA) (J. Arnold and K. L. Pocek, eds.), april 1997, pp. 56–65.

[69] W. Luk, N. Shirazi, S. R. Guo, and P. Y. K. Cheung, *Pipeline morphing and virtual pipelines*, Field-Programmable Logic and Applications, 7th In-

ternational Workshop, FPL'97 (W. Luk, P. Y. K. Cheung, and Manfred Glesne, eds.), LNCS, vol. 1304, Springer-Verlag, Berlin, September 1997, pp. 111–120.

[70] P. Lysaght, *Towards an expert system for à priori estimation of reconfiguration latency in dymamically reconfigurable logic*, Field-Programmable Logic and Applications, 7th International Workshop, FPL'97 (W. Luk, P. Y. K. Cheung, and M. Glesner, eds.), LNCS, vol. 1304, Springer-Verlag, 1997.

[71] P. Lysaght and J. Dunlop, *Dynamic reconfiguration of FPGAs*, More FPGAs: Proceedings of the 1993 International workshop on field-programmable logic and applications (Oxford, England) (W. Moore and W. Luk, eds.), September 1993, pp. 82–94.

[72] P. Lysaght and J. Stockwood, *A simulation tool for dynamically reconfigurable field programmable gate arrays*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems **4** (1996), no. 3, 381–390.

[73] K. K. Maitra, *Cascaded switching networks of two-input flexible cells*, IRE Transactions on Electonic Computers **11** (1962), 136–143.

[74] G. Martin, *Design methodologies for system level IP*, Proceedings of Design, Automation, and Test in Europe (DATE), 1998, pp. 286–302.

[75] G. McGregor and P. Lysaght, *Extending dynamic circuit switching to meet the challenges of new FPGA architectures*, Field-Programmable Logic and Applications, 7th International Workshop, FPL'97 (London, U.K.) (W. Luk, P. Y. K. Cheung, and M. Glesner, eds.), LNCS, vol. 1304, Springer-Verlag, September 1997, pp. 31–40.

[76] ———, *Self controlling dynamic reconfiguration*, Field-Programmable Logic and Applications, 9th International Workshop, FPL'99 (P. Lysaght,

J. Irvine, and R. W. Hartenstein, eds.), LNCS, vol. 1673, Springer-Verlag, Berlin, August/September 1999, pp. 144–154.

[77] N. McKay, T. Melham, K. W. Susanto, and S. Singh, *Dynamic specialization of XC6200 FPGAs by partial evaluation*, IEEE Symposium on FPGAs for Custom Computing Machines (Napa, CA) (Kenneth L. Pocek and Jeffrey Arnold, eds.), IEEE Computer Society Press, April 1998, pp. 308–309.

[78] N. McKay and S. Singh, *Debugging techniques for dynamically reconfigurable hardware*, IEEE Symposium on FPGAs for Custom Computing Machines (Napa, CA) (Kenneth L. Pocek and Jeffrey Arnold, eds.), IEEE Computer Society Press, April 1999, pp. 114–122.

[79] O. Mencer, M. Morf, and M. J. Flynn, *PAM-Blox: High performance FPGA design for adaptive computing*, IEEE Symposium on FPGAs for Custom Computing Machines (Napa, CA) (Kenneth L. Pocek and Jeffrey Arnold, eds.), IEEE Computer Society Press, April 1998, pp. 167–174.

[80] G. J. Milne, *Reconfigurable custom computing as a supercomputer replacement*, Proceedings of the 4th International Conference on High Performance Computing (HiPC) (Bangalore, India), Dec 1997.

[81] R. C. Minnick, *Cutpoint cellular logic*, IEEE Transactions on Electronic Computers **13** (1964), no. 6, 685–698.

[82] ———, *A survey of microcellular research*, **14** (1967), no. 2, 203–241.

[83] K. Nagami, K. Oguri, Tsunemichi Shiozawa, Hideyuki Ito, and Ryusuke Konishi, *Plastic cell architecture: Towards reconfigurable computing for general purpose*, IEEE Symposium on FPGAs for Custom Computing Machines (Napa, CA) (Kenneth L. Pocek and Jeffrey Arnold, eds.), IEEE Computer Society Press, April 1998, pp. 68–77.

[84] S. Nisbet and S. A. Guccione, *The xc6200ds development system*, Field-Programmable Logic and Applications. 7th International Workshop (London, U.K.) (W. Luk, P. Y. K. Cheung, and M. Glesner, eds.), Lecture Notes in Computer Science, vol. 1304, Springer-Verlag, September 1997, pp. 61–68.

[85] C. Patterson, *High Performance DES Encryption in Virtex^{tm} FPGAs using JBits^{tm}*, IEEE Symposium on FPGAs for Custom Computing Machines (Napa, CA) (Kenneth L. Pocek and Jeffrey Arnold, eds.), IEEE Computer Society Press, April 2000, pp. 113–121.

[86] S. F. Reddaway, *DAP – a distributed array processor*, 1st Annual Symposium on Computer Architecture, IEEE, 1973, pp. 61–65.

[87] D. Robinson and P. Lysaght, *Modelling and synthesis of configuration controllers for dynamically reconfigurable logic systems using the DCS CAD framework*, Field-Programmable Logic and Applications, 9th International Workshop, FPL'99 (P. Lysaght, J. Irvine, and R. W. Hartenstein, eds.), LNCS, vol. 1673, Springer-Verlag, Berlin, August/September 1999, pp. 41–50.

[88] M. G. Saleeba, *A self-reconfiguring computer system*, Ph.D. thesis, Monash University, 1998.

[89] S. M. Scalera and J. M. Vázquez, *The design and implementaiton of a context switching fpga*, IEEE Symposium on Field-Programmable Custom Computing Machines (J. M. Arnold and K. L. Pocek, eds.), IEEE Press, 1998.

[90] H. Schmit, *Incremental reconfiguration for pipelined applications*, IEEE Symposium on FPGAs for Custom Computing Machines (Napa, CA) (K. L.

Pocek and J. Arnold, eds.), IEEE Computer Society Press, April 1997, pp. 47–55.

[91] B. Schneier, *Applied cryptography: Protocols, algorithms, and source code in C*, John Wiley and Sons, 1994.

[92] M. Shand, *A Case Study of Algorithm Implementation in Reconfigurable Hardware and Software*, Field Programmable Logic and Applications, 7th International Workshop, FPL'97, LNCS, vol. 1304, Springer-Verlag, 1997, pp. 333–343.

[93] M. Shand and J. Vuillemin, *Fast implementations of RSA cryptography*, 11th IEEE Symposium on COMPUTER ARITHMETIC, 1993.

[94] N. Shirazi, W. Luk, and P. Y. K. Cheung, *Run-time management of dynamically reconfigurable designs*, Field-Programmable Logic and Applications, 8th International Workshop, FPL'98. (R. W. Hartenstein and A. Keevallik, eds.), vol. 1482, Springer-Verlag, Berlin, August/September 1998, pp. 59–68.

[95] R. G. Shoup, *Programmable cellular logic arrays*, Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, March 1970.

[96] R. P. S. Sidhu, A. Mei, and V. K. Prasanna, *Genetic programming using self-reconfigurable FPGAs*, Field-Programmable Logic and Applications, 9th International Workshop, FPL'99 (P. Lysaght, J. Irvine, and R. W. Hartenstein, eds.), LNCS, vol. 1673, Springer-Verlag, Berlin, August/September 1999, pp. 41–50.

[97] ———, *String matching on multicontext FPGAs using self-reconfiguration*, ACM/SIGDA International Symposium on Field Programmable Gate Arrays (Monterey, CA) (S. Kaptanoglu and S. Trimberger, eds.), ACM SIGDA, ACM Press, February 1999, pp. 217–226.

[98] S. Singh and P. Bellec, *Virtual hardware for graphics applications using FPGAs*, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (Napa, CA) (D. A. Buell and K. L. Pocek, eds.), April 1994, pp. 49–58.

[99] S. Singh, J. Hogg, and D. McAuley, *Expressing dynamic reconfiguration by partial evaluation*, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (Napa, CA) (J. Arnold and K. L. Pocek, eds.), April 1996, pp. 188–194.

[100] D. L. Tennenhouse, *Layered multiplexing considered harmful*, H Rudin and R Williamson editors, Protocols for High Speed Networks (1989), Elsevier Science Publishers, IFIP.

[101] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, *A time-multiplexed FPGA*, IEEE symposium on FPGAs for Custom Computing Machines, IEEE Press, 1998, pp. 22–29.

[102] Triscend Corporation, *Triscend e5 configurable processor family*, November 1998, Product Datasheet.

[103] A. M. Turing, *On computable numbers, with an application to the entscheidungsproblem*, Proc. of the London Math. Society **2** (1936), 230–265.

[104] R. Turner, R. Woods, S. Sezer, and J. Heron, *A virtual hardware handler for run-time reconfiguration systems*, IEE Informatics Colloquium on Reconfigurable Systems, Institute of Electrical Engineers, March 1999, pp. 8/1 – 8/5.

[105] M. Vasilko, *DYNASTY: A temporal floorplanning based CAD framework for dynamically reconfigurable logic systems*, Field-Programmable Logic and Applications, 9th International Workshop, FPL'99 (P. Lysaght, J. Irvine,

and R. W. Hartenstein, eds.), LNCS, vol. 1673, Springer-Verlag, Berlin, August/September 1999, pp. 124–133.

[106] M. Vasilko and D. Cabanis, *Improving simulation accuracy in design methodologies for dynamically reconfigurable logic systems*, Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (Napa, CA) (K. L. Pocek and J. M. Arnold, eds.), IEEE Computer Society, IEEE, April 1999, pp. 123–133.

[107] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, *Programmable active memories: Reconfigurable systems come of age*, IEEE Transactions on VLSI Systems **4** (1996), no. 1, 56–69.

[108] S. E. Wahlstrom, *Programmable logic arrays*, Electronics **40** (1967), 90–95.

[109] Xilinx, *The programmable logic data book*, Xilinx Inc, San Jose CA, 1996.