The Reconfigurable Instruction Cell Architecture : Reconfiguration and Interconnects

Ioannis Nousias

A thesis submitted for the degree of Doctor of Philosophy.
The University of Edinburgh.
April 2009

Abstract

A number of architectural paradigms are available that target different needs, either that is programmability or high performance, ranging from GPPs to ASICs. Since no single architecture can deliver both design traits, the current solutions combine one or more of those paradigms to deliver applications that require post fabrication adaptation and high performance. However, such solutions are faced with the added complexity of a multi-disciplinary task. Reconfigurable computing devices promise to bridge the gap between programmable devices and the sheer performance of ASICs, thus offering a more balanced solution.

This thesis offers a comprehensive set of solutions for the implementation of coarse grained, heterogeneous reconfigurable devices, which led to the development of the Reconfigurable Instruction Cell Array (RICA), an architecture template for reconfigurable computing devices. Based on RISC-like instruction cell, this novel design provides a natural flow for compilers. Unique features introduced in this thesis include, a connection oriented configuration encoding scheme, that takes advantage of the intrinsic characteristics of reconfigurable fabrics, along with the decoding process and the interconnects to accommodate this technique. This innovative code-compression technique offers fast reconfiguration speeds. The work continues by describing a method to improve the routability of the design by deriving a suitable physical placement based on the analysis of a set of input data-paths. It also present a performance improvement by reducing the impact of the configuration latency, by pre-fetching the next configuration context and a method to deal with the variable data-path delays, including memory access operations. Finally, this thesis provides an introduction to a Simultaneous Multi-Threading extension for the RICA core. The above are accompanied by a set of mapping tools, such as routing and allocation, down to bit-stream generation and cycle accurate simulator, as well as a set of architecture generation tools, allowing one to deploy different core setups.

Acknowledgements

This thesis would not have been what it is now, without the help of my colleagues and thesis supervisor. More specifically, I would like to thank my colleagues Dr Sami Khawam for his resourcefulness in making the hardware RICA-P1 prototype a reality, Dr Mark J. Milward for his valuable guidance throughout the project, Dr Ying Yi for her work on the compiler and Mr Mark I.R. Muir for his work on the scheduler and software development environment. I would also like to thank Adam Major for his work on porting h264 video decoder on the RICA platform as well as his help in defining the SIMD shuffling operation on RICA-P1. Last but not least, I would like to thank my thesis supervisor Professor Tughrul Arslan for his support and for this unique opportunity to work in an exciting project with important contributions that impact both academia and electronics industry.

Deticated to my father Nikos Nousias(1929-1999)

Contents

List of Figures

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In digital electronics, there are several methods one can use to deliver a particular service or functionality. For instance, an algorithm can be realised using General Purpose Processors (GPP), with or without domain specific extensions, or Field Programmable Gate Arrays (FPGA) or Application Specific Integrated Circuits (ASIC), while the result can be a monolithic device or a synergy of several devices. The suitability of each of those implementation choices depends on the application domain one is working on. There are two main superset domains that sit at the two ends of the spectrum of application domains.

Event Driven Respond to signals received from their environment. They are often referred to as 'real-time' application, meaning they are timing critical and latency intolerant. This is a natural domain for μP using interrupt controllers and performing relatively low computational complexity tasks per event.

Streaming Processing a continuous flow of data. They are performance critical, but latency tolerant. A μP does not have the computational capacity to handle the amount of data and the algorithmic complexity often found in such application domains. DSPs include some well suited computational units, such as multiply-accumulate and VLIWs offer extra degrees of parallelism, which all result in performance improvements. However, when data streams are in the range of 10Gbps and above, VLIWs fall short. One needs to turn over to FPGAs, for those designs that require post-synthesis adaptation. Otherwise, the only solution is an ASIC design.

However, most applications are in fact multi-domain. Parts of the system respond to external events and others process large amounts of streaming data. A digital camera is one example that comes to mind. The camera takes a shot as a response to a user's action, controlling the actuators, zoom lens and flash, while the image signal processing pipe works its way through the raw data received from the sensor. Such a mixture of event driven and streaming application is naturally covered by a combination of programmable and fixed hardware devices.

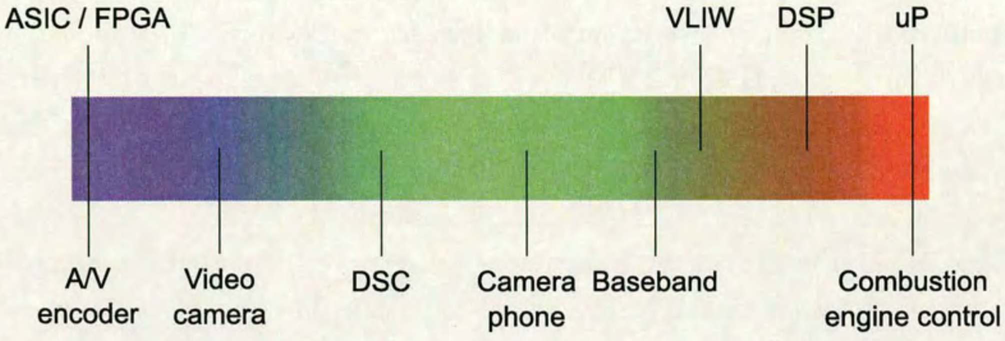


Figure 1.1: Application domain spectrum

Through specialisation, the various architectural paradigms available today position themselves to one or the other side of the spectrum. Unfortunately, that lends itself to various design trade-offs, where no single architecture is able to offer a balanced solution. These are:

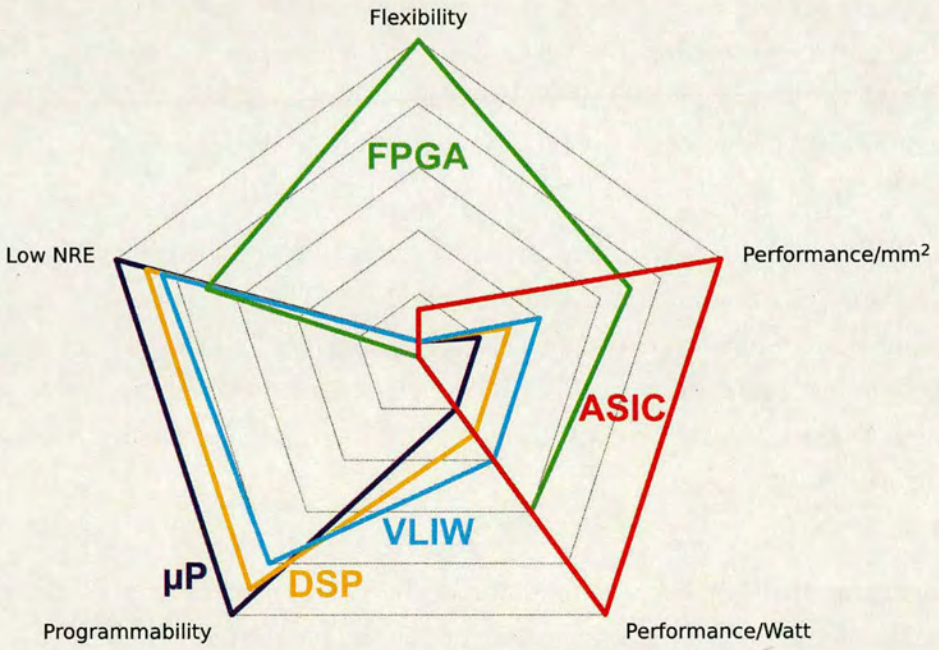


Figure 1.2: Design trade-off differences between uP/DSP/VLIW, FPGA and ASIC

Performance/mm^2 The computational density of the device. Programmable devices such as μP, DSP and VLIW have low computational capacity for their size. The arithmetic units account for a small fraction of the overall design, with register files, caches and the execution pipe logic constituting the majority of the silicon area. An ASIC is theoretically the ideal case, where all silicon is contributing to the computational power of the design, be it transistors or metal layers. In reality, many algorithms have multiple modes of operations and thus parts of the design are inactive. Nonetheless, if looking at the individual functionality, an ASIC is the most efficient solution in the performance/area metric. FPGAs have high computational capacity as well, but they also have significant interconnection overheads that limit their maximum speeds and occupy a significant portion of their silicon area.

Performance/Watt The power efficiency of the design. Similarly to the performance/mm^2 metric, the ASICs represent the ideal case. Each evolutionary step of the programmable device delivers some improvement in this area. Although FPGA are not considered low power devices, their performance/Watt is significantly higher than the programmable devices listed here.

Flexibility The ability to be altered to suit a different environment / configuration. This is what FPGAs are designed for. The programmable devices have fixed set of functionality. Some flexibility may exist on the memory setup, although that's rare. In terms of ASIC designs, this appears as modifiable 'parameters' which can alter some aspects of the design, such as memory geometry or algorithm coefficients.

Programmability Programmability is the way of expressing an algorithm as a set of commands. In essence, a programmable device follows a written 'script', like an actor performing a play. The entry level is an abstract representation of the algorithm which then gets decomposed to a series of intructions supported by the architecture. The advent of programmable devices allowed rapid development of applications.

Low NRE The amount of effort and time required to implement a new design. Programmable architectures have an advantage here, since developing, testing and deploying a new algorithm is relatively easy. The domain specific instructions in DSP may cause some extra effort to utilise them efficiently. Same applies for the ILP present in VLIWs. Both of these depend on compiler technologies to simplify the process. Since the entry level for FPGAs is HDL, the development cycle resembles that of an ASIC. However, the rapid prototyping and testing reduces NRE significantly.

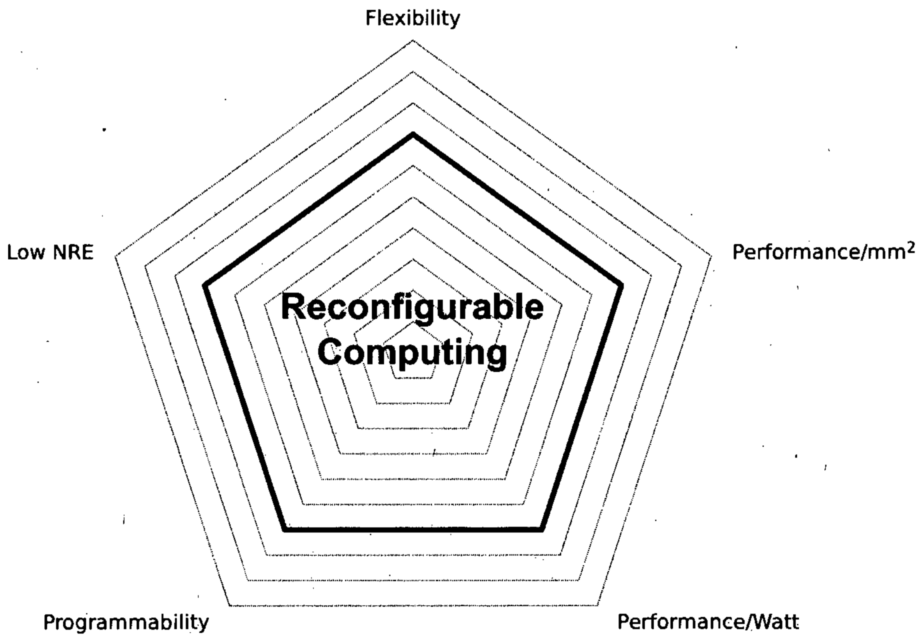


Figure 1.3: The Reconfigurable Computing promise

The graph in 1.2, which is growing outwards, shows that for every design metric, there is at least one architecture type that scores high. Contrary to popular thought, combining two or more architectural types together does not combine the benefits as well. A good example is NRE that will largely depend on the worst performing architecture in that area and could even increase due to the complexity of working on a multi-disciplinary task. A design that can approach the power efficiency and computational density of an ASIC, offering comparable programmability and flexibility to GPPs and FPGAs respectively, is sought by the industry and researched by the academia for nearly 50 years.

Reconfigurable computing promises to deliver a high performance and low power solution that maintains a programmable nature, thus occupying a place between ASICs and GPPs. This is normally achieved using a number of functional units and flexible interconnects, coupled with fast reconfiguration times. Such reconfigurable fabrics are able to deploy hardware data-paths, of the target application, thus improve upon GPPs' limited parallelism.

1.1 The RICA project

The focus in reconfigurable computing has been to reduce the area overhead as well as the reconfiguration times, with varying success. The work presented here describes a novel coarse grained heterogeneous reconfigurable computing architecture, a data-flow machine that delivers high performance, with a low area footprint comparable to small GPPs and significant energy savings, while maintaining a high level software design flow similar to DSPs. This is made possible by employing a number of innovative techniques, which are a direct result of the work presented here.

This thesis is part of a joined EPSRC(GR/S24053/01) funded research project, undertaken at the System Level Integration group (SLIg) lab of the Science and Engineering department of the University of Edinburgh, under the name Reconfigurable Instruction Cell Array/Architecture(RICA). The project came to a conclusion at the end of 2007, having manufactured a technology demonstrator RICA chip, referred to here as the RICA-P1 prototype, filed a number of patents covering key aspects of the technology[1, 2, 3] and a number of publications, including the RICA VLSI journal[4]. The following are the people, other than myself, involved in the development of the main aspects of the RICA architecture and their main responsibilities/contributions within the project, under the guidance and supervision of Professor Tughrul Arslan.

Dr Sami Khawam Joined the project as a research assistant with previous experience in reconfigurable architectures, working in Domain Specific Arrays[5] at the SLIg lab. He completed his doctorate thesis midway the RICA project, including an initial overview of the architecture in his thesis[6]. Sami has been responsible for the majority of the hardware implementation of the RICA-P1 prototype, including the system level aspects of the design, the DMA controller, memory arbiters, debug interface as well as the backend portion of the ASIC design flow.

Dr Mark John Milward Research assistant with a background in parallel data compression techniques. His role included project manager dueties, compiler improvements and sanitising, as well as design verification strategies.

Dr Ying (Maggie) Yi Research assistant with a background in FPGA synthesis tools. Maggie has been responsible for the RICA specific optimisation stages added to the GNU GCC compiler framework, the RICA compiler backend as well as the initial RICA scheduler implementation[7].

Mark Muir A PhD candidate working on instruction scheduling for reconfigurable architectures. He took over the RICA scheduler project, helped define the newer RICA netlist format, an evolutionary step of the format described in Appendix B, and shape the RICA software development environment.

Numerous project have spawn based on the RICA architecture, within the SLIg lab, with a wide variety of applications ported to the RICA platform, ranging from communications standards to multimedia applications. Adam Major's work, on porting an h264 video decoder on RICA, is a notable example that has been used in this thesis as a testing case. Adam also helped define the shuffling SIMD operations on RICA-P1's LOGIC cell, listed in Appendix C.6.

The RICA architecture delivers a number of novel techniques that make it an efficient data-flow machine. The following key aspects of the architecture have been defined in collaboration with the other core members of the RICA project.

Autonomous data-flow machine Unlike most known RCs, the presented architecture is a single fabric capable of handling the flow-control operation as well as the computational data-paths. This further simplifies algorithm development, since the core resembles a typical GPP.

Coarse-grained Instruction Cells Takes a different approach to what granularity and interfaces the functional cells have, compared to other known reconfigurable architectures. By using RISC-like single function instruction cells, with a typical of two inputs and one output, the architecture reflects the internals of a typical compiler, such as GCC and thus simplifies the software development flow.

Distributed registers As part of the heterogeneous nature of the core, the storage elements or registers are distributed inside the array. These distributed registers match the parallel nature of the architecture and provide in place storage, while eliminating the serialised and power hungry access of a register file found in conventional GPPs.

Variable length data-paths A synchronous implementation of a reconfiguration rate controller that determines the duration each configuration context should persist, before the core switches to another. This allows the architecture to dynamically adjust its clock cycle based on the critical path delay of each configuration context.

The following are aspects of the architecture I was responsible for, most of which are covered by specific publications.

Setup agnostic tool-flow The various aspects of the target architecture, such as the array composition and size, physical layout description, as well as abstract description of the configuration context, are all defined in a custom language, referred to here as the RICA netlist (see Appendix B). This internal representation is common to all the tools of the RICA tool-flow, making them architecture setup agnostic and allowing varying levels of details in expressing the configuration contexts, ranging from simple point-to-point description, down to detailed path formations. It is thus easy to deploy diverse array setups, using the same tools, which helps in design explorations.

Refined heterogeneous array A significant portion of the area savings comes from the fact that the architecture is based on heterogeneous single function instruction cells, instead of large ALUs. Using a heterogeneous fabric is a trade-off between area and routability, since allocation is more restricted. This work demonstrates how, with the help of meta-heuristic, a refined placement can be defined that improves routability to levels similar to those of homogeneous counterparts, without requiring extra routing resource.

Parallel data memory access The data memory sub-system of the architecture is partitioned into a number of memory banks, with an arbitration logic that allows parallel access to these banks via the memory interface cells on the core. Simultaneous access to the same bank causes a serialisation with a fixed priority. Due to the parallel nature of the core, the system includes special timer tags per memory interface cell, that aligns the memory access fetches according to the combinatorial delays of the signals feeding the data and address ports of the interfaces. Furthermore, the system provides a suspend and resume mechanism, which dynamically adjusts the reconfiguration rate of the core, taking into account additional memory latencies that can be caused by multiple access to the same bank.

Prefetching A dual page configuration context and the associated fetching mechanism that allows the core to pre-fetch a new configuration, while the previous one is still active. This hides the configuration loading latency in most cases, especially for contexts following kernel operations.

Novel code compression A patented connection-oriented code compression technique, that models the point-to-point paths between functional nodes, achieving high compression ratios. Coupled with a distributed dictionary-based compression and a refined configuration context partitioning, the presented code compression technique offers significant program memory bandwith and latency reductions, which reduce reconfiguration times.

SMT on RCs A patented approach for further enhancing the performance of reconfigurable computing architectures, by introducing Simultaneous Multi-Threading capabilities to the fabric, with partial thread execution.

Chapter 2

Background

2.1 Reconfigurable Computing

Reconfigurable computing is the field of research that attempts to make use of configurable devices, as efficient general purpose computational machines. It was Gerald Estrin who first described, in 1960, a close coupling between a configurable fabric and a general purpose instruction set processor(GPP) as a way to dynamically map circuits on hardware[8]. There have been several architectures proposed in academia and industry since then, such as Morphosys, Garp, Elixent and Pact XAPP [9, 10, 11, 12] to name a few, which follow that same concept. Nowadays, reconfigurable computing is used mainly in High Performance Computing(HPC) application, which make use of a commercially available FPGAs, such as those from Xilinx, and a loosely coupled GPP or a set of GPPs that control them[13].

It's a thin line that separates a programmable from a configurable device, but one that is required to understand their architectural and execution model differences. A 'computer program' is a set of instructions or commands for a given processor. The processor executes those commands in sequence or as otherwise dictated by the control flow of the program, thus forming 'branches' or 'loops'. The most simple form of a processor, a single-issue machine, would execute one instruction at a time. On the other hand, the 'configuration stream' of a configurable device, forms a static data path that implements the entire functionality of the target algorithm. A traditional FPGA falls in this category. Therefore, we could say that a programmable device is dynamic in nature while a configurable one is static, as far as mapping of an algorithm is concerned. However, a new data path can be formed by mapping a new configuration stream to the FPGA, by means of reconfiguration. This is where the two classes of devices fuse together.

A reconfigurable device bridges the gap between static and dynamic mapping. The algorithm is split into a manageable set of operations which are chained together or placed in parallel or both. Each set forms a configuration context that is mapped for a certain duration of time. In other words, the algorithm is mapped both spatially, like in an FPGA, and temporally, like in a processor, thus combining the best features of both worlds. However, such combination introduces a number of difficulties, which make reconfigurable computing an interesting and active field of research.

Rapid reconfiguration is an essential requirement and is one of the biggest challenges faced in reconfigurable computing. The speed of reconfiguration depends on the size of the configuration bit-stream as well as the loading mechanism in place. The configuration size depends on the interconnects complexity and flexibility, as well as the computational granularity of the array. The loading mechanism relates to the coupling between the core fabric and the configuration controller. All these factors, which are covered in more detail later in sections 2.2, 2.3, 2.4 and 2.5, form the basis of a general classification system for reconfigurable computing devices.

Aside from the hardware architecture itself, often overlooked is the importance of appropriate software tools that are essential for programming the system and gaining maximum performance. It is vital for reconfigurable devices to be programmed from a high level language, as this enables complex algorithms to be quickly and effectively mapped to the underlying architecture and thus reduce NRE.

2.2 Interconnections

What allows a reconfigurable computing device (RCD) to form arbitrary datapaths, is the intra-communication network between its functional elements (FE). How flexible and scalable the overall design is, is determined by the design of that network. A number of interconnection network are possible:

Shared Bus A common set of wiring is shared between the functional elements. Communication is established by means of arbitration. An arbitration unit resolves conflicts and assigns the bus to one pair of units at a time, the relationship of which is that of master-slave. The bus is often bidirectional, though the same concept can apply to single-drive unidirectional wiring. Such interconnects can offer high speed burst modes, but suffer from high latencies due to the arbitration process. Memory mapping abstraction is offered, via write and read operation with separate or multiplexed address and data buses.

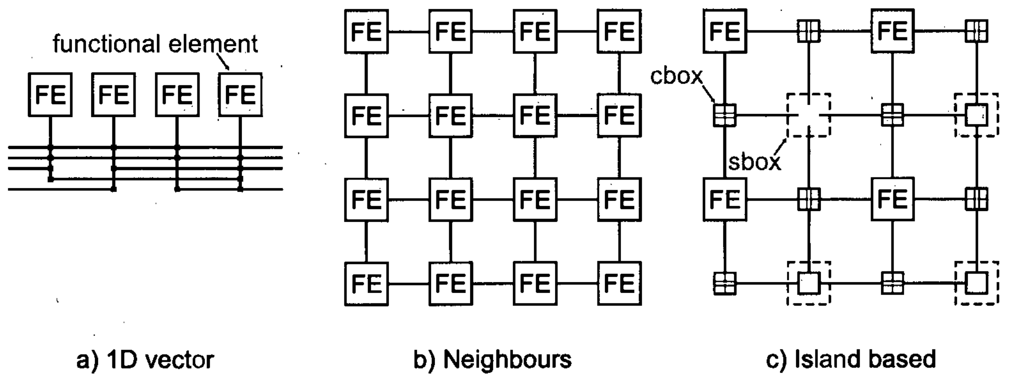


Figure 2.1: Topologies

Standardised interfaces such as AMBA and Wishbone[14] provide easy integration. This type of interconnects does not scale well and the bus becomes a performance bottleneck in high traffic, since all devices share the same routing resources. Extended versions of the typical shared-bus topology include several levels of interconnections, i.e. provide segmentation, in an attempt to localise traffic[15]. This offers some degree of parallelism but with a significant area and complexity overhead. Since most reconfigurable architectures contain tens, hundreds, or million functional elements, as in the case of FPGAs, there is no known practical reconfigurable fabric that makes use of a shared bus topology as part of its intra communication channels. They do make for good system level interconnects where traffic tends to be lower and sporadic. One such example is shown in figure 2.3

One dimensional A set of parallel wires provide a number of 'tap-point' where the functional elements can connect to via connection boxes, as in figure 2.1a. This structure is logically seen as a 1 dimensional vector of units over a set of parallel wires. Both bidirectional and unidirectional wiring can be used. The wiring can be segmented, where some wires extend all the way along the axis, while others may cover only a part of the axis. Each wire segment support one connection at a time and resembles a shared bus. However, unlike shared bus topologies, one dimensional ones often use explicit connection setup and do not rely on arbitration logic. Compared to shared bus topologies, 1D ones offer higher levels of parallelism and thus scale to a higher number of functional units, which can be up to a few tens of units. Since they do not use an arbitration logic they are also simpler to devise, but they are not autonomous. The RaPiD architecture is one such example reconfigurable fabric with a 1D interconnection structure[16].

Neighbour The functional elements of the device, which are often laid out in a lattice structure, maintain local, dedicated, point-to-point channels between their neighbour units, as in figure 2.1b. These are normally single drive unidirectional channels. Data flows on various direction and each 'hop' performs some operation, which could be chosen to feed-through the data to the next unit. Such structure suffer from scalability issues, since larger data-flow graph tend to have multi-fanout edges. An example architecture using this type of interconnects is Garp, which was developed at the University of California in 1997[10]. Others include the multi-core Ambric[17] architecture and the top-level interconnects of SiliconHive[18].

Sea of interconnects A two dimensional mesh of interconnects with switch boxes at the vertical-horizontal crossing points and connection boxes for allowing the functional units to 'tap' onto available channels, as in figure 2.1c. These are logically and historically seen as the next step after 1D interconnects. Again, both bidirectional and single-drive unidirectional wiring can be used. This type of interconnects is termed as 'sea of interconnects' or island based, because the tiled formation of crossing vertical and horizontal wires leaves empty regions where no wiring is present. This is where the functional units are normally placed. This is particularly useful when considering placing memory blocks in those tiles, which often prevent metal layers passing over them. These kind of interconnects scale well to over thousand of unit. The indirect connection between the functional units, introduces delay overheads. For large designs, the island based topology tends to be large and dominate in silicon area. The most familiar representative use of such structure are FPGAs with their island based topologies. Multi-million gate FPGAs have hierarchical interconnects build atop a traditional island-based structure. The Elixent architecture is also based on such island based structure[11] as are other more coarse-grained architecture such as picoChip[19]

Crossbar Each unit can connect to every other unit with a point-to-point connection. Crossbars are fast but do not scale well. In fact the size of the interconnects increases exponentially to the number of units used. An example use of crossbar interconnect is found on the top-level interconnects of SiliconHive and QuickSilver.

Hierarchical Otherwise known us 'clustered' interconnects, these define regions of the interconnection network that are grouped together forming a cluster. A cluster can use any of the above topologies as intra-connection structure. The clusters themselves are connected to each other with a separate interconnection network, which again can be any of the above topologies. Examples include modern multi-million gate FPGAs, SiliconHive with crossbar and nearest neighbour interconnects and QuickSilver with a 2D mesh and crossbar.

2.3 Granularity

The culprit for the excessive routing overhead in FPGAs is the fine-grained granularity of the CLBs, which reflects on both the data and configuration interconnection networks, not to mention the local storage for the actual configuration stream. This is where coarse-grained reconfigurable computing devices (RCD) have an advantage, with significantly simplified interconnects and thus smaller configuration streams. The coarse-grained RCDs perform well in terms of speed and effective silicon utilisation when operating at their native level of granularity, but have a large overhead when diverting from that level.

The graph in figure 2.2 gives an empirical comparison between the area usage for a given arithmetic operation, such as an adder, mapped to an FPGA's CLBs and a coarse grain unit of an RCDs, taking into account the routing resources as well. This figure should be used as an indication rather than accurate data. The inefficiency of FPGAs on coarse grain arithmetic has led

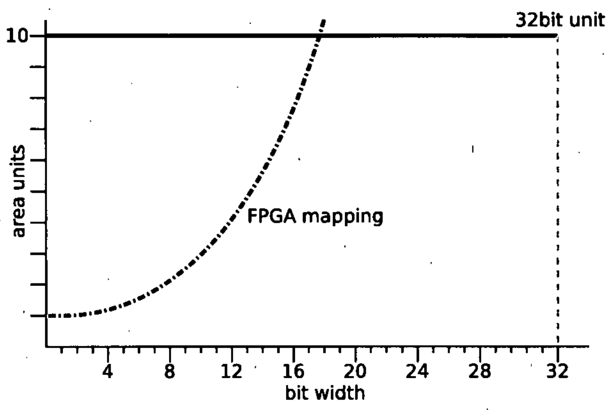


Figure 2.2: Empirical mapping overhead between fine and coarse grained designs

to the design of FPGAs with a number of such coarse grain arithmetic units. Similarly, coarse grain RCDs tend to use bit-level manipulation special units or employ SIMD modes to allow finer grain operation, in a packed form.

Bit Performing bit-level operations. Such fine grain granularity offers high degree of flexibility since it allows to construct arbitrary arithmetic logic of varying width down to the bit level. FPGAs are a classic example of bit level configurable devices. The LUT of the CLBs have a number of 1 bit inputs and a 1 bit output. The Garp architecture operates at a 2-bit level, performing logical or arithmetic operations.

Nibble These perform 4-bit level operations. They are still considered fine-grain. Increasing the granularity normally reduces the overhead of the interconnects, with the expense of a reduction in flexibility. One such example architecture is Elixent[11], which is composed of 4bit wide ALUs.

Byte-Word Coarse grain granularity, performing byte or multi-byte arithmetic and logic operations. This level of granularity is suitable for most programming environments assuming byte operations or above. μP have paved the way for such degree of granularity and high level languages like ANSI C are designed around that assumption. As an example, the Morphosys reconfigurable architectures is composed of 28bit ALUs[9]. Other architecture like SiliconHive[18], combine several VLIWs together to form a fabric of synergetic ALUs. On the end of the spectrum are architectures such as picoChip[19], Ambric[17] and RAW[20], which use a set of small RISC processors connected together via an NoC.

Hybrid Since no given granularity can effectively tackle all domains, combining bit-level with word-level functional units can be a solution. Although beneficial in matching the given granularity of the algorithm natively, the anisotropic nature of such design adds an extra complexity to the interconnects as well as mapping process. Modern FPGAs include wide adders and multipliers, while some also include small RISC cores. Another example is QuickSilver's ACM which mixes bit-level with word-level functional units[21].

2.4 Coupling

The reconfiguration process involves altering the state of the reconfigurable fabric to accommodate new functionality, which often involves a combination of routing between the functional units as well as modes of operation of those units. There is a range of possible scenarios of what makes the decision to reconfigure, as well as what performs the loading process itself. That responsibility is often assigned to the CPU of the system, while the Reconfigurable Processing Unit(RPU) plays a passive role. But that's only one of many possible 'coupling' setups. In most scenarios, where a well defined CPU exist, this can be a single core or multiple cores in a MP environment. Same applies for the RPU. The various broad coupling scenarios are listed below.

Loosely Coupled The RPU is being controlled by the CPU over a communication channel which is shared between the two and possibly other peripheral devices, as in cases 1 and 2 on figure 2.3. Due to the communication overhead, interaction between the CPU and RPU is kept to a minimum. The RPU is configured to operate on a large chunk of data and notify the CPU when it

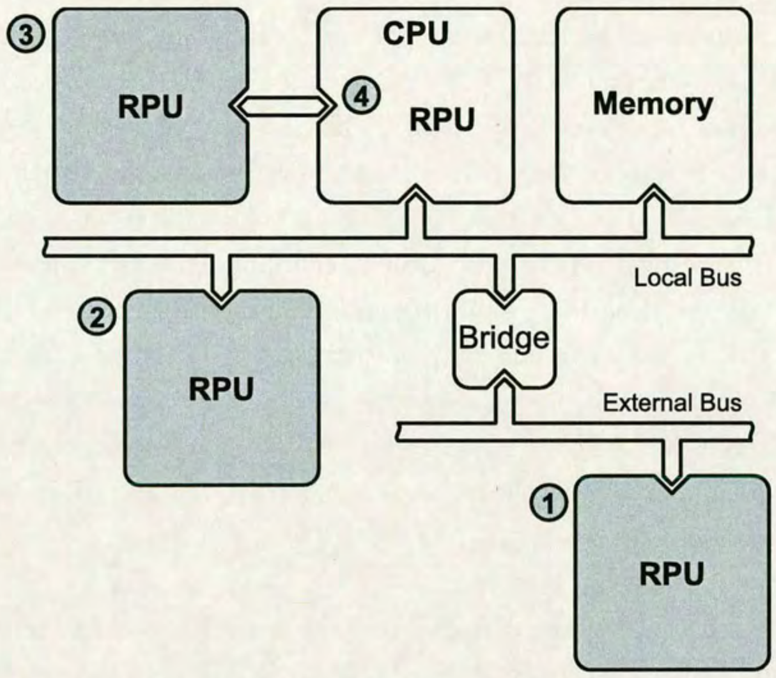


Figure 2.3: Coupling scenarios

is done, via an interrupt, message or having the CPU polling status registers. In such loose coupling, the RPU is a stand alone device. Elixent is an example architecture that falls in this category. It combines a RISC processor with a reconfigurable fabric, communicating via an AMBA bus. Other such architecture is Morphosys, while Cray's XD1 system falls in this category as well.

Close Coupled The RPU is seen as a co-processor unit to the central processing unit, like in case 3 on figure 2.3. The two communicate over a dedicated channel and may share the main memory of the system. The closer coupling allows for more frequent exchange of information between the two. Garp combines a MIPS processor with a reconfigurable fabric which share the same memory subsystem. Pact's XAPP reconfigurable array is also seen as a co-processor to some GPP[12], and so are ADRES[22] and Pleiades[23].

Tightly Coupled The RPU is part of the CPU, as in case 4 on figure 2.3. It is seen as an extension to the CPU's instruction set (ISA extension), able to execute complex, specialised, often SIMD instructions. These are also known as reconfigurable Application Specific Instruction Processors (ASIP). As such, the RPU is part of the execution pipeline and shares registers and I/O interfaces with the host CPU. This form of coupling can offer a more unified programming model and is compatible with compiler frameworks. However, the often very dissimilar timing characteristics between the RPU and the rest of the CPUs instructions, makes scheduling a daunting task. A good example of such coupling is the Stretch architecture[24]. In fact, reconfigurable ASIPs are fairly popular, with applications ranging from Software Defined Radio (SDR)[25] to video processing[26].

Fused/Autonomous The data flow and the control flow are handled by the same fabric. There is no clearly defined CPU. The RPU is autonomous, being able to alter its own configuration based on decisions made from the operations it performs and its current state. The sea of processors architectures, such as picoChip[19], Ambric[17], RAW[20] can loosely fall in this category, while Philips' SiliconHive[18] and QuickSilver's ACM[21] are probably better representatives.

2.5 Programming Model

Since 1960, when the first reconfigurable device was described by G. Estrin, there have been numerous architectures that followed that same idea. Even though they all claimed a new design paradigm shift to be imminent, to date, no such device has received broad acceptance in industry. The reason is that no reconfigurable device has managed to meet all the design targets of the RC promise, as seen in figure 1.3. Some did not quite meet an acceptable computational density, others failed to meet the target power budget, but where most all fail is in programmability. Either the design entry is not convenient or in order to deliver good performance the designer has to tinker with low level constructs. The following lists a number of programming models.

HDL Hardware Description Languages (HDL), such as VHDL or Verilog, describe a design in terms of interconnected gates or higher functional elements. They offer a varying degree of abstraction, being able to describe an algorithm from a Register-Transfer Level up to behaviourally. However, the 'canvas' is the actual silicon, where the designer has to handle timing and synchronisation between the various elements of the design. The design flow involves a number of stages starting from high level, often behavioural level, design, moving on to a gate-level representation and finally a mask-set in case of ASIC. HDLs are the natural design entry for ASICs and as such describe a static design, with any temporal part described explicitly. Architectures that use this design entry lack in programmability due to the very low level nature of HDL. Traditionally, the design entry for FPGAs is HDL.

Assembly The natural representation for intruction set processors is the assembly language, which describe an algorithm as a sequence of operators and their operands and is specific to that processor architecture. Code written in assembly is not easily portable across varying architectures. Assembly is a very low representation and exposes the inner workings of the underlying hardware. When appropriate high level constructs are not available for a given functionality, assembly can be the only way to expose a feature of the underlying architecture. Developing complex algorithm in assembly is a very difficult task and requires good understanding of the underlying hardware. Because of this and because of the portability issue, NRE tends to be high. Most of the architectures will use assembly as means to achieve higher optimisations than it's possible via high level programming.

Custom HLL Most common high level languages, such as C/C++, have been designed with GPPs in mind, which include coarse-grain granularity and temporal mapping, with no explicit notion of parallelism. The spatial nature of reconfigurable devices is alien to those high level languages, which often fail to provide means to express it. This led to most reconfigurable architectures to provide their own high level languages, which are often extensions to existing ones, such as ANSI C. This form of design entry is a good compromise between ease of design and mapping efficiency, but cause inevitable portability issues. QuickSilver's ACM SilverC is one such example, which augments ANSI C with temporal and spatial extensions[21]. The RaPiD architecture comes with its own RaPiD-C language, which as the name implies, is also an extension to ANSI-C. Elixent is another example which makes use of Handel-C. However, Handle-C is closer to HDL than C. Ambric has a mixture of ANSI-C, to program the individual DSP cores and a custom language to describe the parallelism between those core.

Generic HLL The use of common high level languages can improve the chance for an architecture to be accepted commercially. The familiarity to such language as well as the large set of algorithms already written in those languages, makes this form of design entry a desirable one. Abstraction from the underlying hardware and portability are some of the obvious benefits. Architectures such as ADRES and Stretch, recognise these advantages and offer ANSI C/C++ as a design entry. However, software partitioning between the host CPU and the coupled RPU is done manually and thus do not offer a complete solution to the problem. Architectures like picoChip, Ambric and RAW provide ANSI C as a design entry for each of their RISC cores, but coordination is done mostly manually. SiliconHive makes use of ANSI C as well which allows quick prototyping and evaluation. Their compiler is able to extract parallelism from the ANSI C code. For higher efficiency the designer needs to restructure the code to conform to a given style for the compiler to identify parallelism in the design. This design entry is comparable to that of DSPs/VLIWs and thus is seen as an acceptable compromise. Other examples include emerging compiler frameworks that promise 'C-to-gates' design flow, allowing FPGAs to be programmed directly from C, with a certain degree of success[27].

2.6 Code compression

An aspect that is less explored in reconfigurable computing is code compression. A common practise in single issue cores[28], as well as multiple-issue [29] architectures, code compression is applied to reduce the program memory bandwidth as well as the memory storage requirements of those devices.

Code compression is a form of lossless compression, specifically designed around the constraints and requirements of the execution engine of programmable devices. These compression techniques trade compression ratio for low latency and are asymmetric, in that the decoder, which is done in hardware, is much simpler than the encoder, which is performed in software at compile time. Furthermore, unlike general lossless data compressions, code compression techniques require random access to the encoded stream, so that the decoder can quickly respond to the execution flow of the running program.

General lossless data compression starts with modeling the source data based on symbol frequency of occurrence. In essence, what modeling does, is extract the 'information' contained in the data source and thus identify potential redundancy in the code. An 'entropy encoding', such as Huffman, arithmetic, Golomb and Rice coding, can then be used to associate variable size code patterns to input source patterns, according to their probability of occurrence, in ascending order [30], or build a 'dictionary' with the most common patterns [31]. Another option is to use 'run length coding'(RLC) that replace a repeating sequence of symbols with a 'run' and that symbol. The choice of coding depends on the characteristics of the source data, which is identified by the modeling performed at the beginning. In many cases, a combination of the above coding techniques is used to complement each other.

Code compression is architecture as well as application domain dependend. While a dictionary based approach is simpler in hardware and potentially faster than entropy coding in single issue cores, it has been shown that, for multi-issue architectures, under dense program codes, entropy coding methods performs better than dictionary based ones [32]. When it comes to reconfigurable architectures, the configuration stream is mostly sparsely populated and it is natural to expect that an entropy encoding does not scale well to a large number of parallel units. A dictionary compression performs well enough, although for very large number of functional units, such as in FPGAs, the dictionaries can be very large [33].

Code compression can be beneficial to reconfigurable computing devices, due to the high count of functional units such architectures have, which are often underutilised. In many cases, even a simple RLC is enough to remove redundancy in the configuration stream. However, the highly parallel nature of these architecture, which is quite different from that of even VLIWs, makes the design of a fast decoder with low area overhead, as well as the associated program memory interface, a challenging task. Furthermore, the high diversity of reconfigurable computing devices makes it difficult to generalise a code compression technique. The novel technique introduced in chapter 5, represents the first known code-compression technique specifically targeting reconfigurable computing devices, due to its connection-oriented approach, which is an intrinsic characteristic of such designs.

2.7 The role of heuristics in RC architectures

In an ASIC design flow, the high level description of an algorithm, usually expressed in some HDL, is decomposed into primitive operations that correspond to cells from the 'technology library' and are connected with each other so as to form a netlist. This process, which is termed 'synthesis', is followed by a set of physical mapping procedures, with the most notable ones being 'placement' and 'routing'. The placement tries to find optimum relative positions for the cells of the netlist, mainly trying to minimise cross sections and distances, while routing lays metal channels to establish the connections between those cells. Each of those stages are guided by design rules, be it timing constraints, physical spacing, metal layers et cetera. Although the canvas is a plain silicon, both placement and routing operate on some kind of a grid specified by the design rules. There is a lot more to do than that before getting a physical design, some of which are technology process specific, but the above stages form the basis of the ASIC design flow.

The design flow of reconfigurable devices is not very different from that of an ASIC one, at least as far as the above stages are concerned. The application to be mapped is broken down into operations supported by the RC, which are then 'allocated' to the physical instances on the array and finally routed to form the data-paths defined by the target algorithm. The terminology is slightly different, but the stages perform more or less the same functions, with the exception of the compilation phase that in addition to what a synthesis stage does for spatial mapping, it also deals with the temporal nature of the architecture.

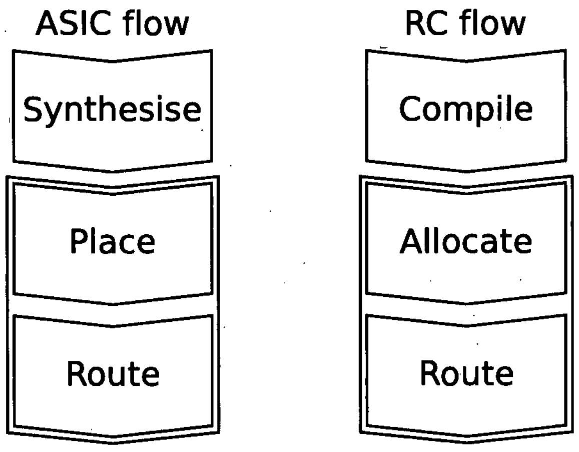


Figure 2.4: Design flow for ASICs and RCs

All these stages are NP-Complete problems, thus the solutions given are approximations. They are usually tackled with some form of heuristic or meta-heuristics algorithm, such as Simulated Annealing(SA), inspired by the process of heating and controlled cooling used in metallurgy to control the crystal structure of metals[34], and Evolutionary Algorithms(EA), inspired by genetics and natural selection process of species evolution[35]. Graph bi-partitioning is also a known technique used in placement algorithms, with Fiduccia-Mattheyses move-based heuristic being a popular one [36].

VPR, which stands for Versatile Place and Route, is an example Place and Route (PnR) tool based on SA[37], delivering quality output and good processing speed. Focusing on FPGAs, it supports homogeneous and heterogeneous FPGAs and means to define the architecture of the interconnects. However, it does make certain assumptions. An island-based topology is assumed for the interconnects and in case of heterogeneous CLBs, they reserve an entire column and can not be freely distributed ac cross the array. Furthermore, there is a strict decoupling of functionality and interconnects.

An example use of GAs for PnR is that used be the the Colt/Stallion architecture[38]. The placement here refers to logical placement or allocation. The chromosome encoding used by Colt's PnR uses integers as genes. As it is usually the case in such implementations, the author makes use of special genetic operators to avoid invalid conditions.

This thesis investigates the various aspects of a practical reconfigurable architecture, such as RICA, in an attempt to provide a fairly comprehensive set of solutions in reconfigurable computing. RICA forms a complete platform, that is build on those techniques described here. It all starts by combining the best features of known design, laying down a good foundation. To ensure a flexible a scalable design, the interconnects are build on an island-based approach. The ultimate goal is to have a programmable architecture and typically such architectures are composed of word-sized functional units. The use of coarse-grained intruction cells is compatible to high level programming languages such as C, which generally does not cope well with bit-wise operations. To further simplify the programming model, the reconfigurable fabric can handle both the flow control and computational data-paths, providing an autonomous solution.

More specifically, chapter 3 gives a general overview of the RICA architecture, followed by a more detailed description of the interconnects in chapter 4, an innovative configuration compression technique in chapter 5 and the configuration loading infrastructure in chapter 6. Chapter 7 presents the RICA toolflow, while chapter 8 introduces a Simultaneous Multi-Threading(SMT) extension for the RICA core. Finally, future directions are given in chapter 9.

Chapter 3

RICA overview

This chapter is a general overview of a complete reconfigurable computing core and its various sub-systems. Its main purpose is to give a birds-eye view of the system, before diving into the implementation details, as well as design decisions that led to this architecture, referred to as Reconfigurable Instruction Cell Architecture (RICA). A first overview of the core architecture can be found in [6] from Sami Khawam, one of the contributors to the design of RICA, others being myself, Mark Milward, Ying Yi and Mark Muir, all members of the System Level Integration group (SLIg) of the University of Edinburgh. A prototype chip has been produced out of this design, which is shown on figure 3.3, and it's used here as a reference. Certain aspects of the architecture, such as the array size, configuration context partitioning, resource mixture and other parameters can be used to tailor the design towards an application domain. The RICA prototype was designed with an MP3 decoder in mind, which helped to define the resource mix and overall size of the core.

A block diagram of the RICA prototype chip is shown in figure 3.1. The 'core' is 32bit wide and is composed of a 9x9 array of 81 switch-boxes and 72 cells out of which 57 are functional cells and the rest are peripheral interface cells and contant value cells. The core is coupled with a 128KByte Data memory and an equally sized embedded program memory. The data memory is split into 4 banks of 8K words each and an arbitration logic allows access of those banks from any of the 4 read and 4 write memory interface cells of the core. The chip was build in UMC 180nm process and occupies a total of $25mm^2$. Figure 3.2 shows the area breakdown of the chip, excluding the I/O pads, where it's shown that the program and data memories constitute 60% of the area, while the core is 40%.

A 'debug interface' is provided that is able to access all major sub-systems of the core, including the program and data memories. The core can be paused and resumed via the debug interface and the output of all cells read from. This allows for a detailed introspection and control of the core via an external source. All accessible parts of the core are memory mapped in the debug interface and can thus be accessed via the address/data pair. A user can issue commands via the debug interface, such as pause and resume and wait for an acknowledgement from the core. The program memory is loaded, as well as

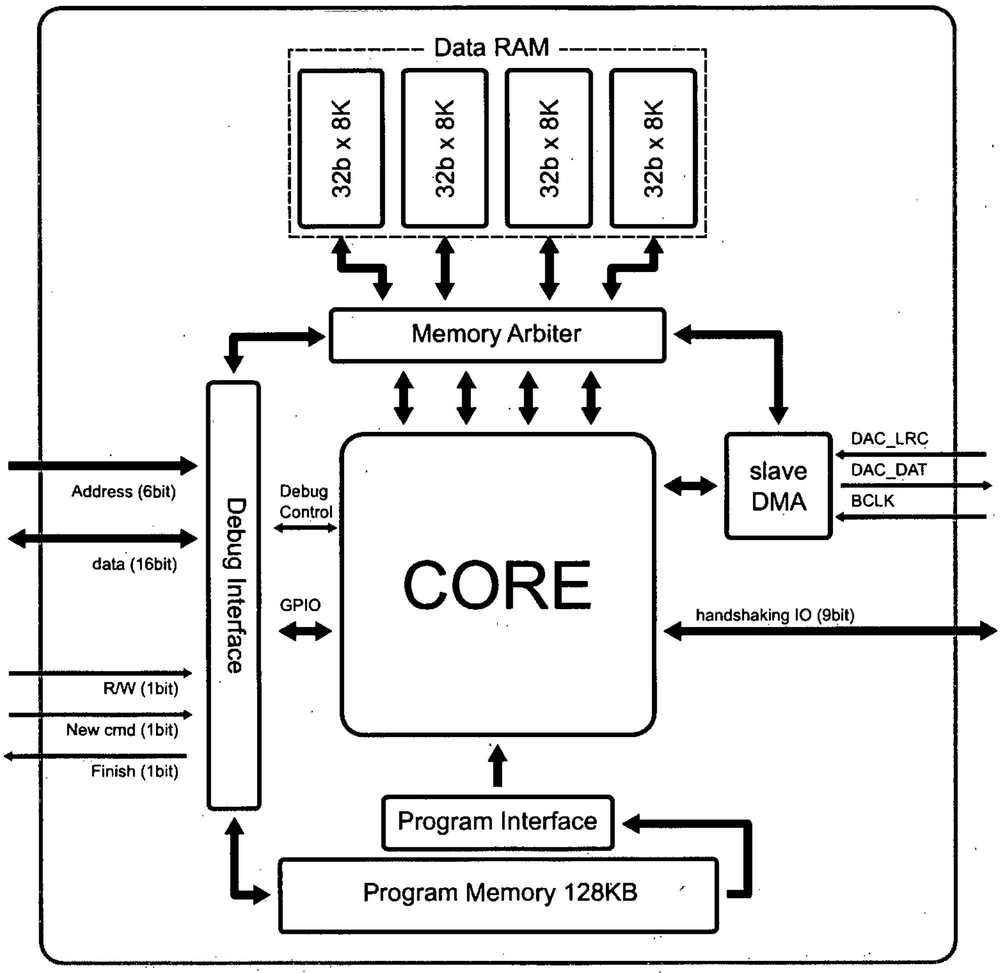


Figure 3.1: RICA prototype block diagram

the data memory is filled with initial data, via the debug interface. The debug interface is flexible enough to allow a host computer to operate the RICA chip as an accelerator unit, although the interface can become a bottleneck if the task is too data-transfer driven.

The RICA prototype includes a 'handshaking protocol' based I/O port, as well as some general purpose I/Os that can be viewed via the debug interface. In addition to those, the prototype include a slave-mode Direct Memory Access(DMA) controller that can be used with a number of Digital to Analogue

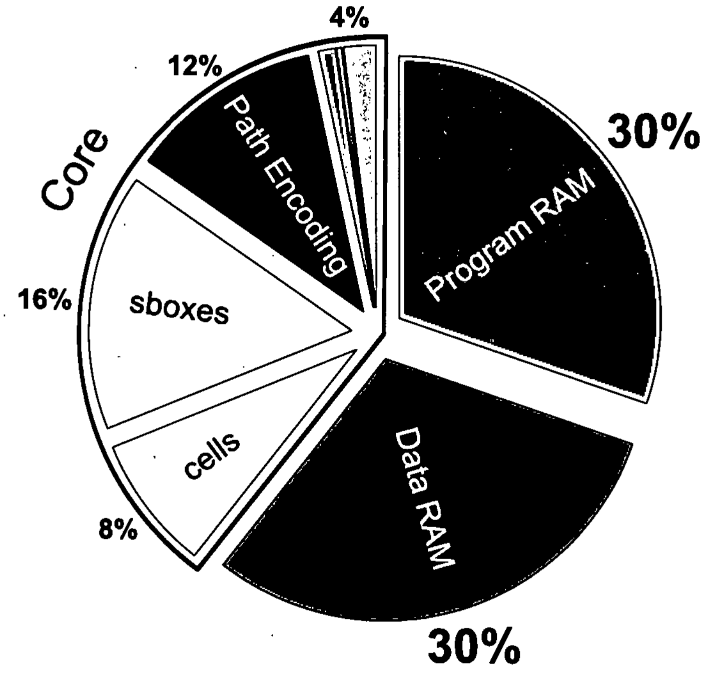


Figure 3.2: RICA prototype area breakdown

Converters(DAC) operating as a master. This has been specifically added to allow a direct audio output from the chip, since MP3 decoding was the primary target function. The DMA controller hooks onto the memory arbiter and is seen as another memory interface cell. It can thus access the data memory whilst the core is running, since conflicts are automatically dealt with by the arbitration logic. The implementation of the debug interface as well as the DMA controller, are not part of this work.

Figure 3.3 shows an actual photo of the RICA prototype chip, with watermarks indicating the program and data memories as well as the core in the middle. The visible metal layers over the core area are the toroidal channels.

The following sections give a brief overview of the interconnects and reconfiguration sub-system, which allow the core to dynamically deploy computational data-paths in few cycles, as well as the general programming model and the data-memory sub-system of the implemented RICA prototype design. More details are provided in subsequent chapters.

data-paths that are then mapped to the core. In fact, the choice of computational units, in terms of granularity and operation, has been such that it's easy to do such mappings. RISC operations are known to be compiler friendly [40].

The end result is a set of 'netlists' of interconnected RICA instruction cells, which form the final configuration 'contexts' that runs on the core. These are often refereed to as 'steps' in RICA terminology. The contexts execute in sequence or as otherwise is dictated by the control flow graph(CFG) of the program that was mapped. The execution flow is controlled by a branch unit in a similar way to conventional processors, by updating the program counter accordingly. This branch or flow control unit is nothing more than an instruction cell inside the core. The branch condition is evaluated by the core's cells, within that context, and is routed to the input of the flow control, just as any other connection between cells. Thus, the branch conditions form an integral part of the cores execution, unlike coupling scenarios to a host CPU, where the branch conditions form a separate program running on the CPU.

Unlike conventional processors that execute one instruction at a time, or in case of VLIWs, several in parallel, RICA can form any arbitrary combination of operations, by chaining together several computational units as well as having several such data-paths independent from each other. Therefore, where VLIWs suffer from ILP limitations, RICA has the flexibility to run dependent and independent data-paths at the same time. Furthermore, the ability to chain operations together allows things like pipelining, by inserting pipeline registers in the data-paths and perform re-timing as well as other ASIC like optimisations.

Compared to DSPs and VLIWs, which rely on complex instruction caches to reduce the penalty of instruction fetches from main memory in case of computational loops, RICA is able to map the operations of such loops as a static context and thus form an ASIC-like implementation of it, assuming enough resources are available. For example, a functional loop with 10 instructions that iterates $1K$ times, will result in one instruction fetch in RICA. In contrast, a single-issue conventional processor needs $10K$ fetches from its level 1 cache, while a multi-issue core would still need around $2-3K$ fetches due to a typical ILP limitation of 4 parallel instructions [41]. Fewer memory accesses to the program store will normally translate in lower energy consumption on that part.

3.2 Heterogeneous versus Homogeneous

Table 3.1 shows the post routing area figures for the various functional cells in the RICA prototype chip. It also includes the area of an ALU with the compined capabilities of each of the individual primitive cells, excluding the 32bit multiplier. The table does not include the various interface cells, such as the general purpose and handshaking I/Os as well as the control flow unit and DMA interface cell. In a homogeneous environment, these are either assumed to be accessed by all ALUs or remain as separate cells. Eitherway, the impact these have on area is relatively small. At the end of the table are the total area figures of the total 56 heterogeneous functional cells in the RICA prototype chip and the same for an array made of ALUs. This shows a dramatic difference in area between the two design choices, with the heterogeneous one occupying a mere $1.3mm^2$, while the ALU based being $23mm^2$ in size.

	Area (mm^2)	Count
64bit Mul	0.253	2
32bit Mul	0.161	2
Adder	0.028	5
Logic	0.028	3
Shift	0.045	3
Mux	0.0028	3
Comp	0.0088	2
WMEM	0.0043	4
RMEM	0.004	4
Reg	0.0017	31
ALU	0.403	57
Heterogeneous	1.30	57(combined)
Homogeneous	23.01	57

Table 3.1: Heterogeneous versus Homogeneous

It's important to note that these two designs do not offer the same degree of flexibility. However, what a heterogeneous design offers is the choice to trade flexibility for area, based on the requirements of the target application domain. Apart from the reduced flexibility in terms of functional resources, a heterogeneous architecture also impact the routing flexibility of the design, due to restricted allocation. Section 7.6.1 in chapter 7 shows that, with the help of meta-heuristics, a carefully chosen physical placement can compensate for that loss of allocation choices, improving overall routability of the heterogeneous design to comparable levels to that of a homogeneous one.

3.3 Interconnects

RICA prototype's interconnects are a two dimension toroidal mesh with single-drive, unidirectional channels, composed of switch-boxes with 4-to-1 multiplexers. Each output multiplexor of an sbox is driven by the input ports of the other sides and the remaining fourth input is driven by the output of a 'terminal' cell. Each terminal cell has two input ports which are driven, indirectly via a multiplexor playing the role of a connection-box, by the outputs of the nearby switch-boxes. This sort of topology is similar to an 'island-based' one found in FPGAs

The total configuration of a switch-box (sbox) and associated connection boxes (cbox), is thus $cfg_{width} = 6 \times 2 = 12bit$. The RICA prototype has 81 such sboxes, giving a total of 972bits raw configuration for the interconnects. However, as it is discussed in chapter 5, the interconnection configuration is not stored in its raw form and is rather compressed with a special encoding scheme. More details on the architectural decisions around the interconnects are provided in section 4.1.

3.4 Reconfiguration

The reconfiguration sub-system is responsible for delivering the configuration to the interconnects and cells as well. It is closely related to the configuration

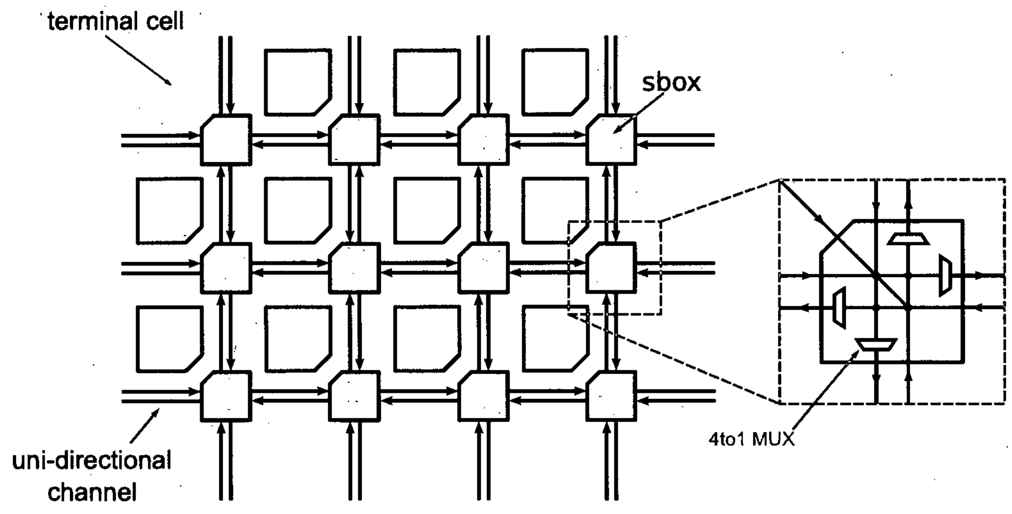


Figure 3.5: RICA prototype interconnects

format and the program memory interface and thus design decisions on one part influence the others.

To reduce the configuration size, the RICA prototype utilise a special compression scheme that is designed around path formation. Furthermore, to reduce the size of the configuration distribution network as well as the program memory interface, the core is divided into a set of partitions, each of which is able to accept a configuration fragment and deliver it to one of its members. The program memory interface reflects that same partitioning. The partition is also referred to as 'group', while each member of that group is referred to as an 'insertion point' (IP) when talking about configuration loading. An insertion point can been considered to refer to a cell for now, although in reality it's not a one to one mapping.

The number of partitions depends on the size of the array as well as the size of the individual configuration fragment. The size of the configuration fragment is set by the path-encoding, which is described in chapter 5, while the size of the array is set by the target computational power of the device. The RICA prototype, which is a 9x9 array, has 10 such partitions, resulting in a manageable program memory interface width of just under 256bits. Since a configuration context is larger than the interface width, it is split into a set of rows, each of which has fully or partially populated partitions. Figure 3.6, shows a 6-way partitioning of a configuration context, which is also split into multiple rows.

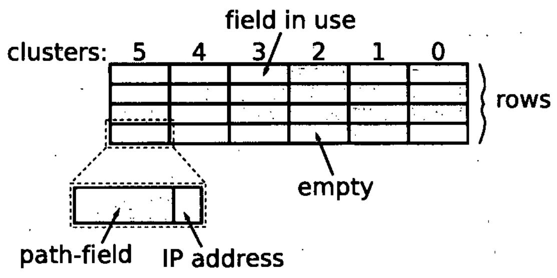


Figure 3.6: Configuration partitioning

Each configuration fragment targeting a group, includes the encoded path description and the address of the insertion point (IP) to be delivered within that group. A decode logic is present on the interconnects, which unfolds and distributes the configuration across the core. More details are found in chapters 5 and 6.

3.5 Data memory arbiters

One of the most explored topics in computer science is that of memory architectures and most notably the work that has been done in *caches* [42]. Caches are used to hide the latency of slow memory interface, usually an *off-chip* one, by introducing one or more layers of faster but smaller intermediate buffers that shadow the main memory. In case of multiple core or processor configurations, cache coherency systems are used to ensure all local copies of the data are in sync and up to date [43].

The choice of a memory architecture is highly architecture dependent. A single processor will issue one *load* or *store* operation per cycle. A single issue memory sub-system is enough to handle such access patterns. Ideally the low level cache is able to serve requests at the same speed as the processor issues them. The processor's data bandwidth is often dictated by the cache's memory bandwidth.

As seen at the beginning of this chapter, a single RICA configuration context can map several data-paths of various operations at the same time, including memory access operations. This raises the need for a multiple issue memory system. In RICA, this is done by including multiple memory interface units for read and write requests as part of the array. The implementation in use is based on a multiple memory bank system which shares the same address space using arbitration logic. Sharing the address space simplifies the programming interface of the RICA core.

The memory interfaces are divided into two sets, one for writting and the other for reading. Each memory interface has its own data and address ports, as well as a convenient offset port. The memory interface units are able to form parts of the operation-chains in data-paths. Memory reads are issued during the execution of a step and the core dynamically adjusts the required expiration time of that step, depending on the total memory latency due to arbitration. This is explained in more details in section 6.2. Memory writes happen at the end of a step and can progress while the core continues executing, as long as there is no other memory activity present.

3.6 SIMD instruction cells

The RICA prototype supports packed arithmetic or otherwise known as Single Instruction Multiple Data(SIMD) [44] or vector arithmetic. In a first spin of the RICA instruction set, as seen in [6] Appendix A, each cell has 3 modes of operation, 'single', 'half' and 'quarter' integer arithmetic and more specifically, for the 32-bit RICA core, these are 32-bit, 16-bit and 8-bit operations respectively. Each mode performs a single computation of the right bit-width. Supporting packed arithmetic is a natural progression. Each computational cell is able to perform a full single integer, or 2 half integer, or 4 quarter integer operations, as listed on table 3.2.

The benefits of vector operations are easy to spot. However, there is a hidden cost which is often overlooked. Apart from simple, straight forward vector computations, like the ones in figure 3.7a, most algorithms need to operate on the elements of a resulting vector, like in figure 3.7b. This requires unpacking

	operation width	operation count
SI	Single Integer (32-bit)	1
HI	Half Integer (16-bit)	2
QI	Quarter Integer(8-bit)	4

Table 3.2: RICA cell operation modes

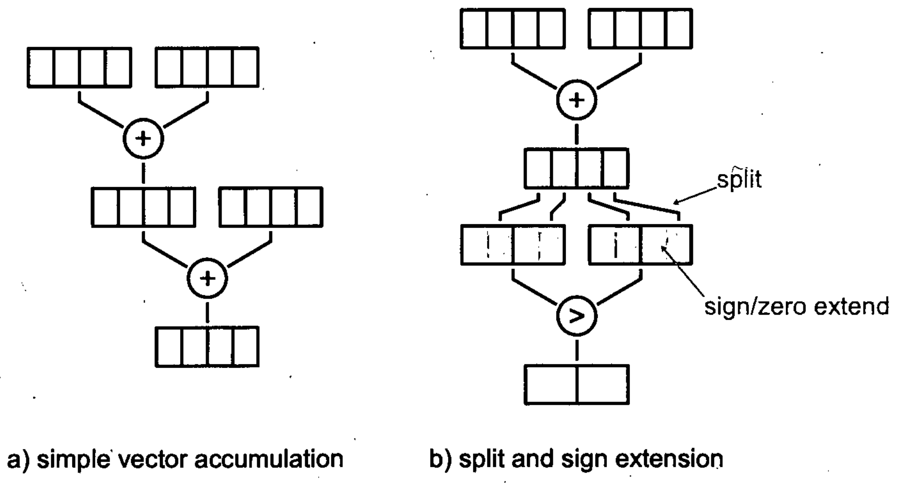


Figure 3.7: Example vector operations

and often re-packing the data in different bit-widths, which in case of upwards conversion, it also involves sign or zero extension. The RICA prototype includes special modes of operation of the LOGIC cell, which perform packing, unpacking and shuffling operations, to accommodate SIMD arithmetic. A full list of the supported SIMD modes of operation for the LOGIC cell, as well as the other functional cell, can be found in Appendix C.

3.7 Performance figures

The RICA prototype chip was tested using the libmad MP3 decoding library and compared to ARM7 and ARM9 cores. Before looking at the results, it's important to mention some of the design decisions made before fabrication to reduce risk and meet the fabrication dates. Some of those design decisions impact the overall area of the chip and others the performance. These are covered briefly below. The core is fully synthesisable and no layout optimisations have been performed. This first RICA prototype serves as a technology demonstrator more than anything else and is an incredible achievement considering it is the first chip we ever produced.

Minimum step time of 2 cycles The RRC design was forced to a minimum of 2 cycles due to combinatorial loops reported by the layout synthesis. A redesign of the RRC unit would have resulted missing the deadlines. This limits the shorted configuration context time to 2 master clock cycles.

Data RAM reading takes several cycles The available memories at the time required several cycles to perform a read. This limits the memory bandwidth of the core

Two cycles for Path Decoding As discussed in chapter 5, the configuration stream is encoded with a novel technique named 'path encoding'. Post-layout simulations revealed timing errors on the decoder, which forced an extra cycle to be inserted to make sure the decoder will work. Unfortunately, there was not enough time to redesign the decoder.This impacts the performance of the core.

No clock-gating Although, part of the design spec, clock gating was not implemented in time for fabrication. Instead, the clock tree feeding the registers inside the core is live all the time. Clock gating would have reduce power consumption by a large margin.

Very relaxed routing The core utilisation was set to about 50%, to simplify place and route. This impacts the total core area. Normal figures are in the range of 80 to 90% utilisation. It's worth mentioning that there is no intrinsic problem achieving such utilisation.

Other than those shortcomings added by design, there were a few errors discovered in the final hardware after testing. These are listed here:

Redundant program RAM reads The core performs an extra program memory fetch at the start of the step. This does not affect the core's operation, however it consumes power.

Steps with no RMEMs have 1 extra delay cycle Part of the design specification, a time tag of 0 means 1 cycle for RMEM cells that are not cascaded. The idea is that, the inputs of an RMEM cell would require at least 1 cycle from the beginning of a step until they are ready, so there is no point in counting that. This feature increases the available range by 1 cycle for those cells. However, a bug in the hardware ended up making all RMEM tags to be set to a +1 cycle. Combined with the fact that the minimum step time is 2 cycles, that ends up having a minimum of 3 cycles per step. For more information on RMEM cells, refer to section 6.2 in chapter 6.

The achieved maximum frequency for the RICA setup is 36MHz. This clock frequency is obtained by the PCI bus clock of the FPGA board that the RICA board is attached to, for interfacing with the host PC. The core might be able to be clocked as high as 50MHz, but this was not tested, since such clock generator was not available. It's important to note that the RICA prototype does not include a PLL.

Taking all of the above issues into account, a revised version of the RICA prototype chip was estimated and the results are included here to reflect the performance figures that could have been achieved had those errors and design shortcomings been avoided. Table 3.3 lists the results obtained by running libmad MP3 decoder on the ARM cores and the RICA prototype and revised version. All cores run at 36MHz, the maximum frequency of the RICA prototype setup.

Figure 3.8 depicts the performance ratios betwen the various cores, obtained by the results of table 3.3.

The results show that, performance wise, even the unmodified RICA prototype chip is faster than the ARM7 cores by 2.3x and by 1.8x from the ARM9 core. The revised one improves on those figures with 4.25x and 3.3x respectively. Area wise, the core of the RICA prototype chip is 3x larger than ARM9, while the revised one is 2x. On the energy front, the unmodified prototype

	Core Area (mm^2)	Time per frame ($msec$)	Core energy per frame (uJ)
ARM7-TDMIS	0.62	63.70	642.00
ARM7EJSC	1.25	63.70	710.79
ARM9966ES	2.00	49.50	1249.00
RICA P1	6.00	27.50	1093.00
Revised RICA P1	4.00	15.00	491.00

Table 3.3: ARM vs RICA prototype

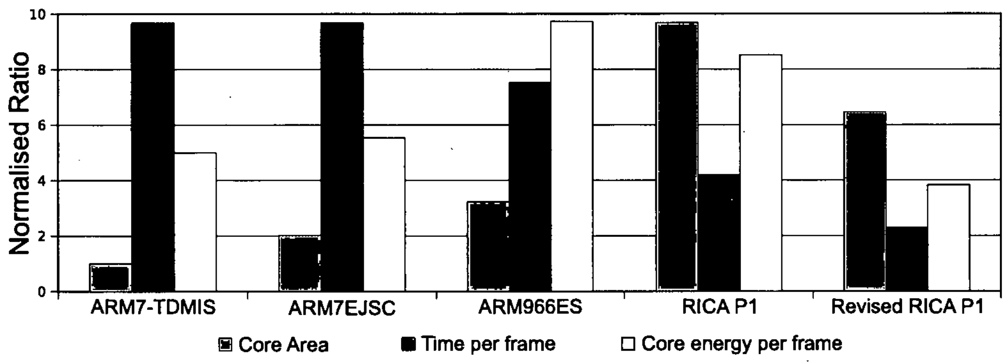


Figure 3.8: ARM vs RICA normalised performance ratios

performs slightly better than the ARM9 core. However, looking at the revised version shows the true potential of the architecture, with a healthy 2.5x lower energy consumption compare to ARM9.

The above results show that, despite being an early technology demonstrator and a purely synthesisable design, the first RICA prototype chip performs favourably compared to ARM9, even with the design errors that ended up landing on the final hardware. Furthermore, considering that the core has a regular structure, with the repeatable instances of switch boxes, as well as the low number of cell types, silicon optimisations can be easily applied to further improve area figure as well as delay values. Any small improvements in the switch boxes can have a big impact in the overall design, which is why FPGA vendors, like Xilinx, handcraft their interconnects.

Chapter 4

Interconnects

Probably the most important aspect of a reconfigurable architecture is its interconnects. Their ability to adapt to different configurations is a direct function of the flexibility of their interconnects and the prime factor of differentiating them from static ASICs. However, this added flexibility comes with a significant area overhead, making it their biggest strength and their greatest weakness at the same time. Most of the design trade-offs come down to the right mixture of flexibility and tailoring of the interconnects.

4.1 Topologies

In a typical FPGA implementation, demonstrated in [45], over 70% of the silicon estate is used by the various data interconnection components, with only 10% being the actual logic and the remaining 20% forming parts of the configuration distribution network. This is why significant design efforts, in commercial FPGAs, is put in optimising those interconnects by handcrafting them and making the right trade-offs between flexibility and area overhead. Despite the clever interconnection topologies employed to maintain high routability, designs mapped on FPGAs are more often than not routing limited.

Tailoring of the interconnects can provide the right amount of flexibility for a given area budget. This tailoring usually refers to the number of available routing tracks per channels and the topology in use. Figure 4.1, shows the basis of the interconnects on FPGAs. Termed as 'island based' interconnects, these are a class of 2D interconnects where wiring channels do not intersect with the logic units. The result is a 'sea' of interconnects surrounding the logic units, hence the metaphor.

The interconnects are composed of 'connection boxes' (C) and 'switch boxes' (S). The connection boxes offer 'tapping' points for the logic units on the interconnects, while the switch boxes provide cross sections between the two dimensional grid lines of the interconnects. The interconnects are clearly decoupled from the computational part, allowing them to be handcrafted independently, which is usually the case. Multi-million gate FPGAs have segmented tracks that span across multiple switch boxes, offering bypass channels and thus reducing the number of 'hops' required to connect CLBs together, thus forming

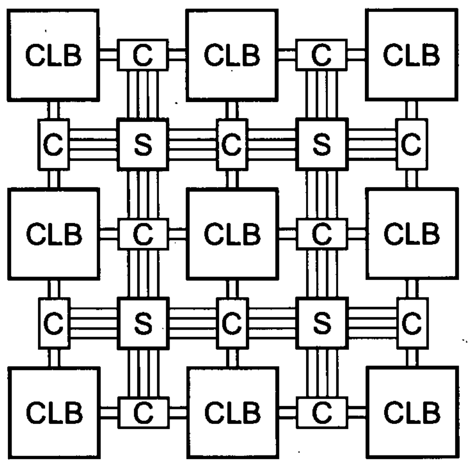


Figure 4.1: Island-based interconnects

a complex hierarchical structure[46]. Traditionally, the FPGA interconnects have been based on bidirectional channels with tristated buffers at the cross sections, but recent designs have moved to unidirectional and single driver channels, due to the higher stability they offer compared to tristate buffers, in modern technology nodes, such as 65nm[47].

RCs make use of similar interconnects to FPGAs, albeit being simpler and smaller. Figure 4.2 shows an overview of the interconnects used by the RICA prototype as in section 3.3. This topology is similar to the island based on seen in figure 4.1, but with toroidal channels, which can be seen as bypass channels.

The main difference here, compared to the traditional island-based interconnects, is that of the 'tap points' of the logic unit. In figure 4.1, the logic unit taps on the intra-connection channels between the switch boxes that surround it, while in figure 4.2 the logic unit taps on the intra-channels of one of the switch boxes with its neighbours. This is better shown in figure 4.3.

At first, this seems to complicate wiring, however it lends itself to a different implementation style, where a switch box and an attached logic unit, can be self contained. This simplifies the automation of core generation.

A more minimal sbox design is that shown in figure 4.4. These 'T' shaped

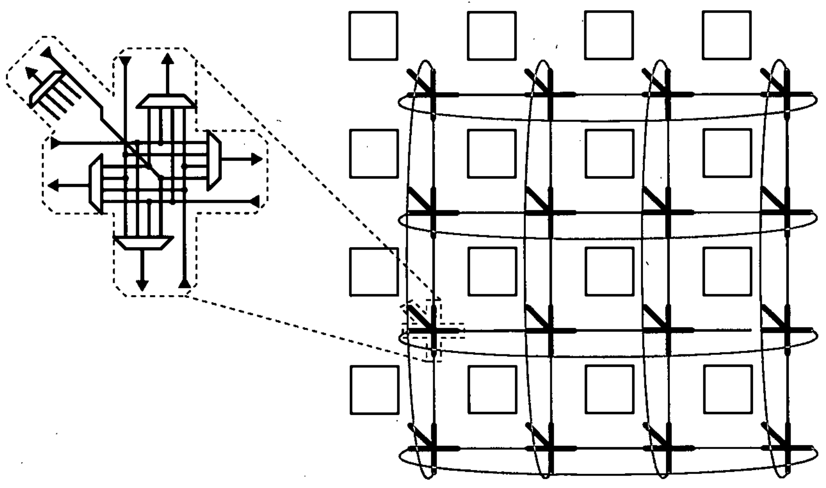


Figure 4.2: Cross switch-boxes

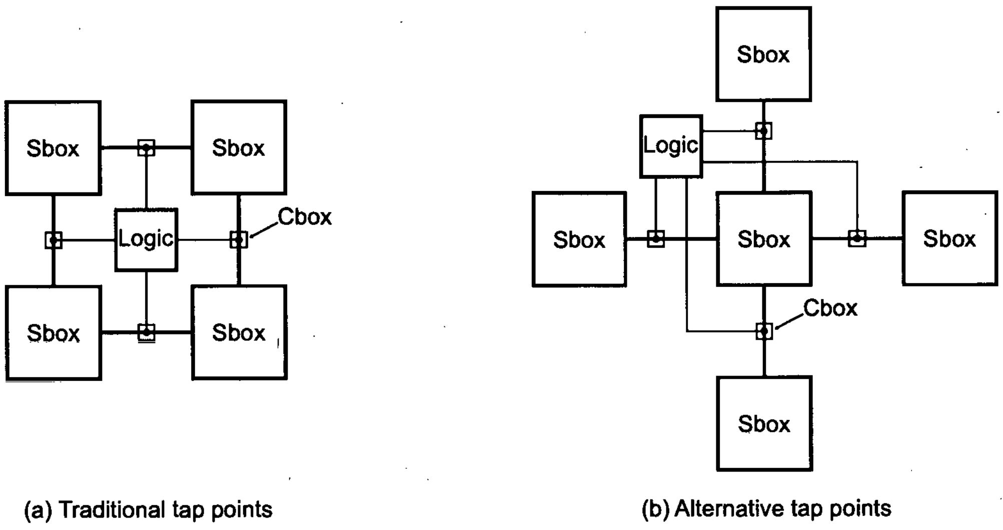


Figure 4.3: Logic unit tap points

sboxes are composed of 2to1 multiplexers, unlike the 4to1 ones in the 5-way sbox. Three T-sboxes are needed to form similar connectivity to that of a 5-way sbox. The major benefit of a T-sbox is the lower fan-out count. Every 4to1 multiplexer on the 5-way sbox has 6 fan-outs (including the 2 terminal inputs), while a T-sbox has only 3. This has the potential for a smaller design with comparable routability and average wire-length.

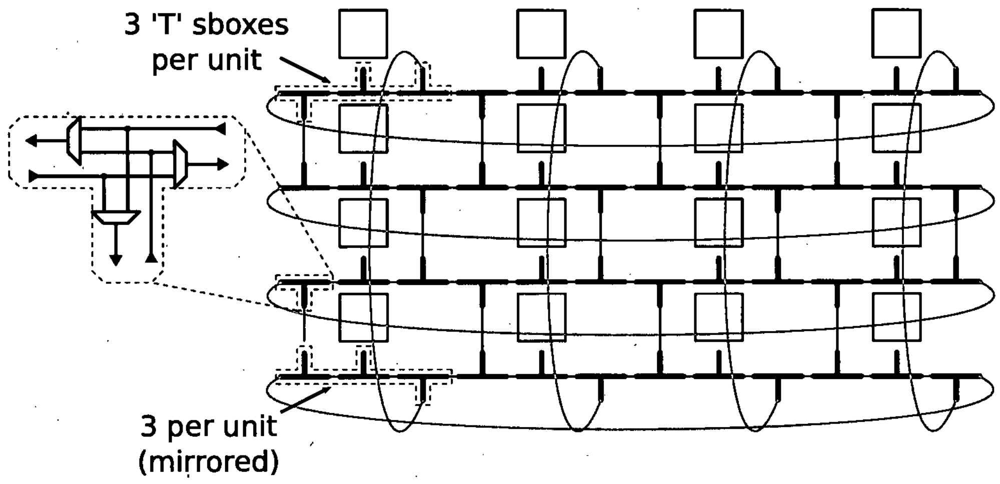


Figure 4.4: T switch-boxes

4.2 Passive or dynamic interconnects

The interconnects on an FPGA are explicitly set by the configuration bit-stream, during the loading phase. These interconnects provide no dynamic or run-time adaptation, while the routing path-ways are pre-computed during the mapping stage. Such interconnects play a 'passive' role in the reconfiguration cycle. This simplifies the design of the interconnects, however leaves no room for adaptability.

Dynamic interconnects, such as message-passing based ones, offer an extra level of flexibility with the expense of some latency, which can be non-deterministic at times. These dynamic interconnects have many things in common to computer networks, which is why they are also known as Networks on Chip(NoC). A NoC can alter its connection routes at run time, allowing, among other things, the dynamic reallocation of operations to functional units. Dynamic reallocation is useful in fault tolerant applications, as well as Simultaneous Multi-Threading SMT, as seen in chapter 8.

The remaining of this chapter includes an investigation in switching and routing techniques used in dynamic interconnects, leading to the proposal of a lightweights switching technique and the inspiration to the on-line routing described in section 6.1.2.

4.3 Switching and routing

Dynamic interconnects can be classified by their 'switching' and routing techniques. There are several switching techniques available, including circuit switching, packet switching and wormhole switching along with several derivatives of those, such as store and forward and virtual cut-through. Wormhole switching is one of the most popular switching techniques for NoW and NoC for its simplicity and small footprint.. Additionally, due to the pipelined nature of this technique, the propagation delay in wormhole switching is practically independent of the routing path [48].

Adaptive and deterministic (or oblivious) are the main available routing techniques. Both have their strengths and weaknesses. However deterministic is most popular in NoC for its simplicity. Performance wise, adaptive routing provides better results for high workload, while deterministic is best for low workload[49].

The implementation simplicity of wormhole switching and its adequate performance for relatively low traffic workload makes it the preferred switching technique for most application from Network of Workstation to Networks on Chip [50]. One of the main advantages of wormhole switching is a nearly buffer-less design. A packet is forwarded in a pipelined fashion flit by flit and the routing decision is made on reception of the header flit, while the data flits simply follow the header flit.

Unfortunately wormhole switching suffers from contention problems. A blocked header flit results in a sequence of blocked data flits that tie a number of routing channels. In some cases this can lead to dead-locks, where two or more packets are blocked by each other in a vicious circle.

As a solution to reduce congestion, prevent dead-locks and improve wormhole switching techniques performance, the physical channels are often time sliced into a number of independent logical channels. These logical channels are also known as virtual channels (VC). However, the performance improvements of VCs come with a price. VCs increase input/output buffer requirements by a large factor, which contradicts the initial decision for a wormhole switching technique. For a large number of VCs the performance benefits are overshadowed by the resulting bandwidth degradation. However this is less of an issue in NoC environment where area and power limitations are dominant. A practical number of VCs is two per physical channel for moderate packet sizes.

Conventional implementations for VCs use a set of circular queues, of a depth equal to the maximum expected packet length, one for each VC. Thus the maximum required buffer size per router for such implementations is as follows.

$$b_{vc} = vc \times plength \times ports \tag{4.1}$$

Where vc is the number of VCs, $plength$ is the maximum packet length and $ports$ the number of ports per router. In [51] the authors propose an improvement to the above, making the total buffer size independent of the number of VCs, i.e $b_{vc} = plength \times ports$. Further reduction of the required buffer size is achieved by using special flow control mechanisms. A number of choices are available including buffer-less, FIFO based, credit-based [52] and handshaking based as mentioned in [53].

The lightweight wormhole switching technique with virtual channels, proposed here, is based on a simple credit-based, rate-control flow control mechanism, suitable for NoC environments.

4.3.1 Virtual Channels with dynamic rate control

This section describes a proposed light-weight flit rate control technique, referred to as Virtual Channels using Dynamic Rate Control(VCDRC), which attempts to minimise intermediate buffer requirement, by controlling the online traffic and adjusting the sources flit transfer rate using feedback signalling. In the event of contention, the receiving node changes each packets transfer rate so that the accumulative rate matches that of the physical channels one. It does so by requesting each source node to adapt their transfer rate accordingly. This signalling eventually propagates to the source logic unit.

Based on the proposed scheme the required buffer size does not depend on the number of virtual channels and packet size, but only depends on the number of ports per router. The following equation gives the theoretical maximum required buffer size per router.

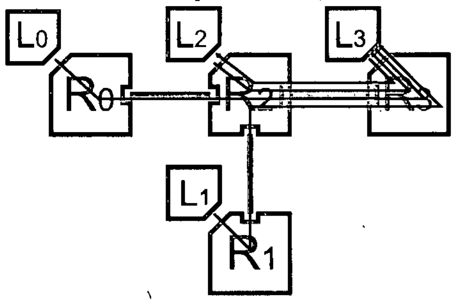
$$b_{vcdrc} = 2 \times (ports - 1) \times ports \tag{4.2}$$

The above equation derives form a logical analysis of the implementation. Each output port accepts flits from all input ports except its pair input, since loop-back is not allowed. Assuming pipelined packet flow, which stands true for a wormhole switching based network, the output buffering requires a storage of 2 flits for every associated input port. A more analytical explanation on the factor of 2 is given below in relation to table 4.2. Thus based on equations 4.1 and 4.2 the buffer size ratio between the VCDRC technique and the conventional implementation of VCs is given by the following equation.

$$b_{vcdrc} = b_{vc} \frac{2 \times (ports - 1)}{vc \times plength} \tag{4.3}$$

For a typical router of 5 ports using 2 virtual channels and a packet length of 128 flits, the VCDRC technique requires just 3% of the intermediate buffer size of the conventional VC implementation or 6.25% when compared to the one proposed in[51].

Output buffering tends to result in complex implementations compared to input buffering. Nevertheless, the intrinsic characteristics of the VCDRC technique are better suited for an output buffering approach, since direct control of the output flit rate is required. VCDRC is based on a 'suspend and resume' approach, where packet flits are temporarily suspended in an attempt to balance the transfer rate according to on-line network traffic. Using feedback flow control signals the routers make sure that the incoming flit rate matches the achievable output flit rate of the physical channel. The following description assumes a flit equal to a phit (physical channel width), although the proposed technique applies for other data partitioning ratios. VCDRC is built on top of a wormhole switching network, which means that both data and flow control are pipelined.



$L_0 \rightarrow R_0 \rightarrow R_2 \rightarrow R_3 \rightarrow L_3$
$L_0 \rightarrow R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow L_3$
$L_2 \rightarrow R_2 \rightarrow R_3 \rightarrow L_3$
$L_3 \rightarrow R_3 \rightarrow R_2 \rightarrow L_2$

Figure 4.5: Simple communication scenario

A small data transfer example is shown on figure 4.5 for a comparison between a conventional virtual channels implementation and the proposed one. The example illustrates how multiple packets share the same physical channel (here R2 to R3 channels). The logic units are designated as L_x while the routers are R_x. The example assumes two unidirectional channels per link. Table 4.1 presents the communication pattern at a flit transaction abstraction level in a conventional VC implementation

Each row shows the output channel of a router/logic unit connected to its neighboring unit. Each column represents one instant in time. The shaded numbers are packet IDs and for simplicity are equal to their source logic units index. Thus packet 0 has generated from logic unit L_0. The buffer utilisation table shows the total buffer size in flits required by each router in every instant. The point of intersection, router R_2, is highlighted.

At t=1, router R_2 receives 2 flits simultaneously, one from R_0 and one from R_1, to be routed on the same physical channel. Since the incoming rate is higher than the achievable output flit rate of 1flit/cycle, the remaining flits are buffered at R_0 routers output buffer. From t=1 to t=5 router R_0 accepts 2 flits and outputs 1flit every clock cycle, which means that a buffer of at least the size of one packet is required. Nevertheless, in order to support the worst

	output ports (packet IDs)																		
L[0]	0	0	0	0	0	0	0	0											
R[0]																			
		0	0	0	0	0	0	0	0										
L[1]	1	1	1	1	1	1	1												
R[1]																			
		1	1	1	1	1	1	1											
L[2]								2	2	2	2								
R[2]								3	3	3	3	3	3	3	3	3	3	3	3
			0	1	0	1	0	1	0	1	2	0	1	2	0	1	2	0	1
L[3]							3	3	3	3	3	3	3	3	3	3	3	3	3
R[3]																			
				0	1	0	1	0	1	0	1	2	0	1	2	0	1	2	0
time	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
	total buffer utilization (in flits)																		
R[0]	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
R[1]	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
R[2]	**0**	**0**	**2**	**3**	**4**	**5**	**6**	**8**	**10**	**11**	**11**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**
R[3]	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 4.1: Communication pattern in conventional VCs

case scenario, when the output channel is busy, based on equation 4.1, the router R_2 would require 192flits total buffering for 3 VCs and 16 flit packets. Here the total buffer utilisation is 11flits, which results in a total of 6% buffer efficiency.

Table 4.2 shows the communication pattern in VCDRC, using the same traffic scenario of figure 4.5.

The output flit pattern of router R_2 towards R_3 remains the same as in the previous example. What changes here is the output of the source routers and logic units. Two time instances are highlighted here, time t=2 and time t=8. At t=2, router R_2 signals back to routers R_0 and R_0, requesting them to reduce their flit rate, by sending a *transmission/idle* ratio and a starting point delay (designated as S_x). The starting point causes a misalignment in time between the two sources, which prevents them from overlapping. The decision of which one goes first can be made based on packets priority tags.

The transmission ratio defines a variety of transmit and wait output stages, for example 2 wait and 3 transmit. Here the routers start with a wait stage

	output ports (packet IDs)																		
L[0]	0	0	0	0	S_0	S_1	S_2		0		S_0	S_1	0			0			0
R[0]																			
		0	0	S_0		0		0		S_0	0			0			0		
L[1]	1	1	1	1	S_0	S_1	S_2	S_3			1			1			1		
R[1]																			
		1	1	S_0	S_1		1		1	S_0	S_1	1			1			1	
L[2]								$\boxed{2}$	$\boxed{2}$							$\boxed{2}$			$\boxed{2}$
R[2]								$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$
			0	1	0	1	0	1	0	1	$\boxed{2}$	0	1	$\boxed{2}$	0	1	$\boxed{2}$	0	1
L[3]							$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$
R[3]																			
				0	1	0	1	0	1	0	1	$\boxed{2}$	0	1	$\boxed{2}$	0	1	$\boxed{2}$	0
time	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
	total buffer utilization (in flits)																		
R[0]	0	1	1	1	2	2	1	1	0	0	1	0	0	1	0	0	1	0	1
R[1]	0	1	1	2	2	2	2	1	1	0	0	1	0	0	1	0	0	1	0
R[2]	**0**	**0**	**2**	**3**	**2**	**1**	**1**	**2**	**3**	**4**	**4**	**3**	**3**	**3**	**2**	**2**	**2**	**2**	**2**
R[3]	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 4.2: Communication pattern in VCDRC

(blank box). The same happens at t=8, when R_2 receives packet 2 to be routed on the same channel. All transfer rates are updated accordingly. Packet 3 is routed via a different unidirectional physical track, so it does not interfere with the rest of the packets.

Since feedback signalling requires 1 cycle to take place, the router needs to store 2 flits from each packet. Thus if the physical channel was busy at the time of reception at t=3, buffer utilisation would be 4flits. This is where the factor of 2 comes in equation 4.2.

4.3.2 VCDRC performance

As described above, the proposed technique is based on packet handling using their IDs. Each packet is assigned a temporal slot which is identified by its ID. From a hardware implementation point of view, this would require the use of Content Addressable Memory (CAM). CAMs are normally larger than normal SRAMS and more specifically a UMC-13 CAM is around 2.5 times larger than an SRAM with the same technology and configuration. This factor is taken into account on the following performance analysis for a more fair comparison with the conventional VC implementation.

SystemC was chosen as the system level language for the implementation of a Transaction Level Model (TLM) of the proposed technique and the conventional VC implementation, for performance analysis. Using a custom made script the simulation model allows the realisation of any given topology and can be tested under various communication patters. Figure 4.6 shows the percentage of activity distribution on a 4x4 2D-mesh network using uniform traffic. Additionally, it reveals internal properties of the selected traffic pattern, which can be used in conjunction with the performance results that follow.

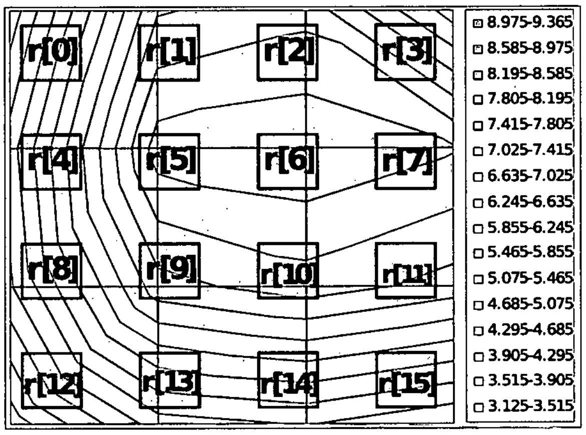


Figure 4.6: Percentage of activity distribution on a 4x4 2D-mesh network under the tested traffic pattern

A multi-hop Direct Network 2D-mesh topology is used here as a motivational example. The proposed technique can be applied to other multi-hop or complete DNs, such as hypercubes or toroïdal topologies with similar results. All tests were made using the same uniform traffic pattern as seen in 4.6 over a 100Kflits. For comparison reason the buffer area for the conventional VC implementation was made equal to that of the proposed technique. This figure was obtained by rearranging equation 4.3 while taking into account the 2.5 area factor explained at the beginning of section 4.3.2. The following expression provides the allowed packet length when both implementations have the same buffer area, for a given number of VC and port count.

$$plength = \frac{5 \times (ports - 1)}{vc} \tag{4.4}$$

Based on equation 4.4, in order to support 2 virtual channels, the packet length should be 10 flits for a 5-port router. It should be noted that a 10flit packet is relatively small and that in practice, packets of 64 or higher lengths are used.

The graph in 4.7 shows the average throughput achieved on a per packet basis, for a range of packet injection ratios on the conventional VC implementation (light-line) and on the proposed technique (dark-line). The graph shows that the performance benefits of VCDRC increase as the packet injection ratio increases. More specifically at a 0.02p/c the performance difference is 7%. For extremely high packet injection ratio (above 0.2p/c, which is not shown here) both implementations saturate around 0.2flits/cycle. The maximum possible injection ratio for this test case of 10flit packets and 16 nodes is 1.6p/c.

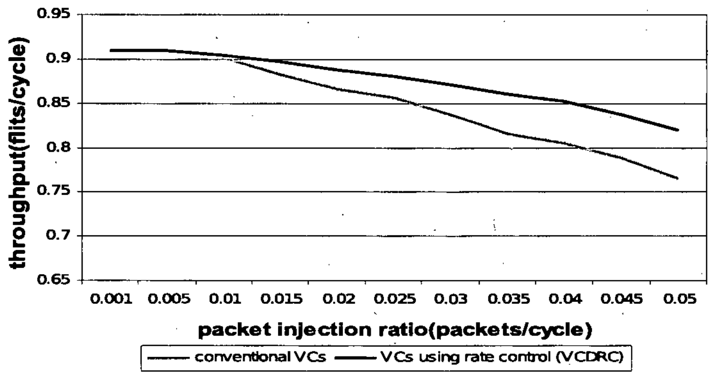


Figure 4.7: Average throughput in flits/cc per packet in relation to packet injection ratio.

The graph in 4.8 illustrates the average latency in clock cycles on a per flit basis for both implementations. Again VCDRC delivers better results, especially for high packet injection ratio, i.e. high traffic work load. It is worth mentioning that the performance benefits are even more apparent for larger packets. For a typical packet length of 64flits the performance difference is up to 15% for a 0.02p/c packet injection ratio. Of course for a 64flit packet length the buffer requirement for the conventional VC implementation increases by 6 times, while for VCDRC it remains unchanged.

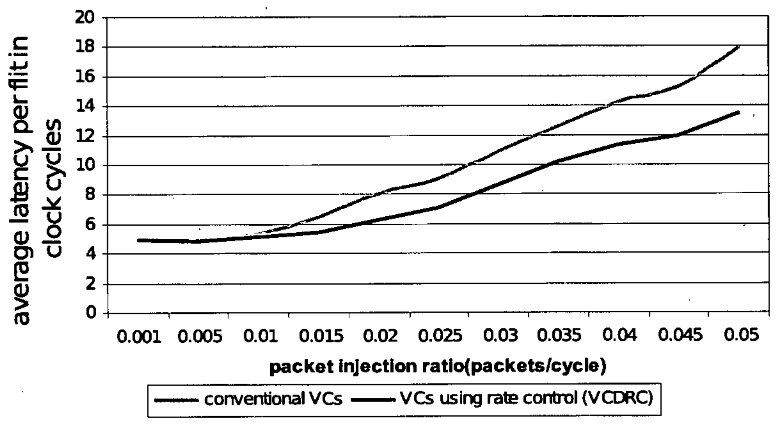


Figure 4.8: Average latency in ccs for each flit

Although the VCDRC technique does offer a measurable improvement over conventional VCs in terms of performance/area, the fact remains that such a technique adds latency, area and complexity to the overall design. However, even though a wormhole routing technique may not be the most appropriate solution for the main interconnects of a reconfigurable system, as described in section 5, the packet based path establishing approach can be used as part of the configuration loading infrastructure, to reduce the bandwidth requirements.

Chapter 5

Path Encoding

5.1 A connection oriented approach

The configuration of the interconnection network of a dynamically reconfigurable architecture, such as RICA, is probably the most critical section of the design. The speed of reconfiguration as well as the bit efficiency of the configuration stream depend on it. The interconnects configuration describes the various connections between the computational, storage or interface units, as dictated by the data-path mapped on the array. Such a description can vary from an explicit set of raw configuration bits controlling the various routing nodes, up to a high level description of the participating connections.

This chapter describes a novel connection-oriented encoding of the interconnects configuration, which exploits intrinsic characteristics of a valid path, referred to as *path encoding*. To assist in explaining the path-encoding scheme the following sections make use of a 2D-mesh lattice. The technique applies to toroidal topologies as well as any other form of multi-hop direct network topologies. In the following examples a 'node' is a routing unit within the network, such as an s-box (switch box), while an 'edge' is a unidirectional link/wire that connects two nodes. A 'path' is the chain of edges and nodes that form a multi-hop connection from the source to the destination node.

The path-encoding reduces the amount of information required to represent a point-to-point multi-hop connection. As such it can also be seen as a compression algorithm specifically applied to this domain. Most importantly the path-encoding offers an abstract representation of a path that, as shown below, can reduce the overhead of routing establishment or 'rendering', on a direct network topology [54]

5.2 Path-Encoding Basics

The path-encoding scheme exploits the intrinsic characteristics of a valid point to point connection on a given interconnection lattice. Between two points in the lattice there is a finite number of possible paths that can form a point-to-point (multi-hop) connection. Some of these paths are practically useless, such as those that form loops. Others are too inefficient to be considered a valuable solution, such as those that form obscure mazes. Finally there is a subset of paths that are relatively short and are fine candidates for a selected solution.

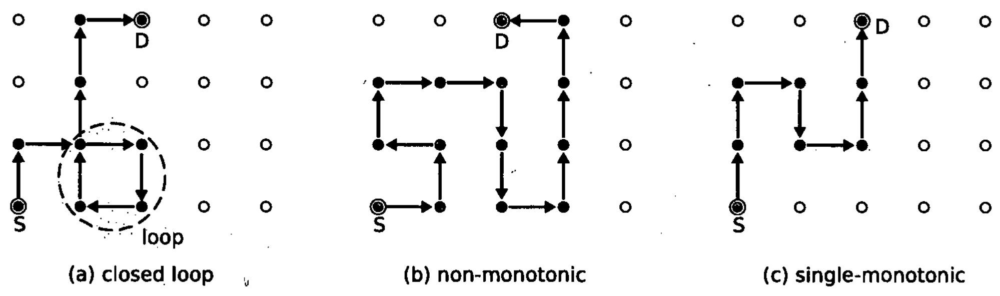


Figure 5.1: Path formation scenarios

Out of these 3 cases only the last 2 form valid paths. The second case, even though valid, covers a large set of interconnection resources and thus forms a very long and inefficient path. The third case is only a subset of the available valid solutions. This case is characterised by its *single monotonic* style, in one of the two axes. A *single monotonic* path is one that maintains a given direction on one of the two axes, in contrast to a *fully monotonic* one that both axes are maintaining a given direction. The later ones also define a set of paths with optimum length for a given directional choice. Note that in a multi-dimensional environment only one dimension of the path needs to be monotonic to be considered a single monotonic path.

5.3 Path Encoding displacement format

Any given point-to-point connection on a multi-hop network has at least two nodes that are fixed, the *source* and the *destination* node. The chain of participating intermediate nodes defines the final path of the connection. Looking at a single monotonic path, there are a number of intermediate nodes that are 'forced' to participate to ensure a continuous path. These nodes are named here *implicit nodes.* The rest of the nodes that play an important role to the formation of the path are called *explicit nodes.* Similarly an edge that emerges from an implicit node is also called *implicit edge.* Edges that emerge from *explicit nodes* are called explicit as well or as discussed below, *controllable segments.*

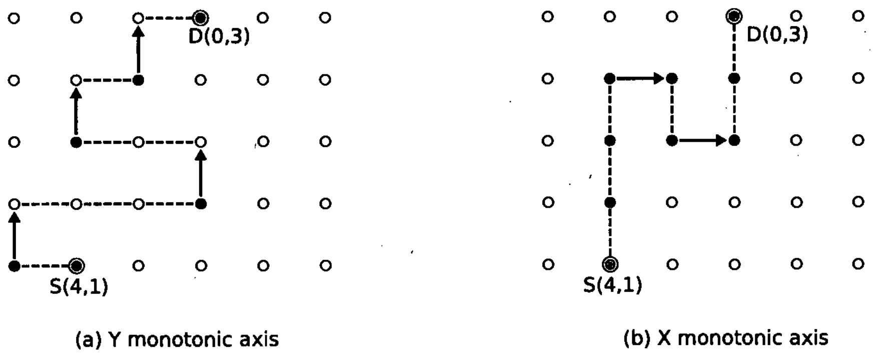


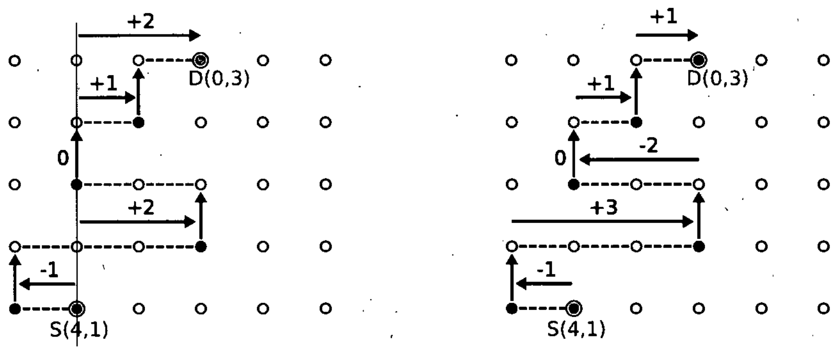
Figure 5.2: Y or X monotonic axis

The above figure shows two example single monotonic paths for the same point-to-point connection, one on the *y-axis* and one on the *x-axis.* Explicit nodes are shown as solid dots while explicit edges are shown as arrows.

As seen above, in order to form a single-monotonic path the only information required is the location of the source, the relative position of the explicit nodes or edges (controllable edges), plus the direction of the path on the monotonic axis. In Figure 5.2 on the y-axis monotonic example, there are 4 controllable edges. Each of these controllable edges can be placed in any position within a row. Using as a reference the sources x-coordinate, the position of each controllable edge can be expressed as the displacement from the reference line.

Based on the above observation the example path can be expressed as two flags, one that defines the monotonic axis (Y or X) and one for the path direction (positive/negative), one coordinate and a vector of 5 signed integers. The above set of information forms the path-encoding packet and for a 2D-mesh path is as follows:

$$[\mathtt{m}] \quad [\mathtt{dir}] \quad [\mathtt{S}(\mathtt{y_s},\mathtt{x_s})] \quad [\mathtt{size}] \quad [\mathtt{d_0}\ldots\mathtt{d_{n-1}}] \tag{5.1}$$



(a) reference line displacements **(b) differential displacements**

Figure 5.3: Controllable edges' displacements

$\boldsymbol{m}$ is the monotonic field, a flag that specifies the monotonic axis (X or Y)

$\boldsymbol{dir}$ is the direction in the monotonic axis.

$\boldsymbol{n}$ is the number of displacements.

$\boldsymbol{S(y_s, x_s)}$ is the source field and uses $log_2(Y) + log_2(X)$ bits (rounded up to the nearest integer), where Y and X are the arrays dimensions.

$\boldsymbol{size}$ is the size of the displacement vector.

$\boldsymbol{d_0 \dots d_{n-1}}$ is the displacements field and uses $n \cdot log_2(Y)$ or $n \cdot log_2(X)$ bits, depending the monotonic axis

Based on the example path on Figure 5.3a, using the source node S as a reference, the path-encoding packet would be as follows:

$$[0] \quad [1] \quad [4,1] \quad [5] \quad [-1,+3,-2,+1,+1] \tag{5.2}$$

Or for the example in Figure 5.3b using differential displacements this is:

$$[0] \quad [1] \quad [4,1] \quad [5] \quad [-1,+2,0,+1,+1] \tag{5.3}$$

These assume the upper left corner node has a coordinate of $(0,0)$, that a 0 flag is used for a Y and 1 for X monotonic-axis and positive path direction is 0 while negative is 1.

5.4 Non-monotonic Path-Encoding extension

A non monotonic path-encoding requires a way to alter the reference axis. In a monotonic environment, a y-monotonic path uses an x-reference axis and vice versa. Further more the encoding must ensure valid path formation, in other words, avoid loops or conflicts. The example on figure 5.4 adds an extension to the monotonic path-encoding that allows non-monotonic paths to be formed.

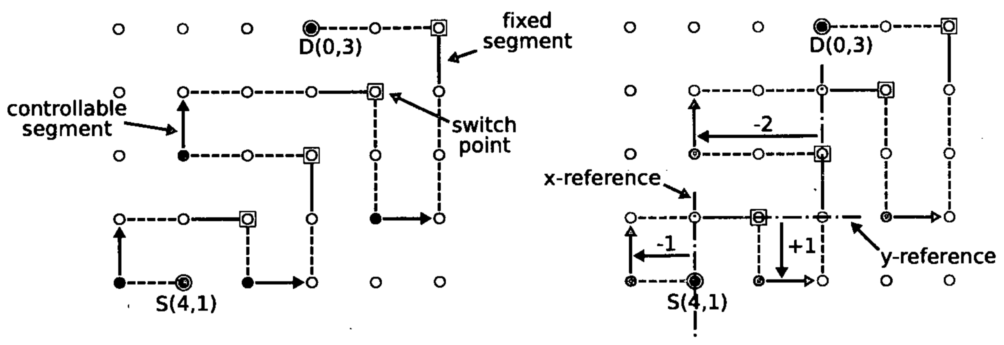


Figure 5.4: Non-monotonic path-encoding extension

The added field in this scheme is able to alter the reference axis, which is needed to device a non-monotonic extension of the path-encoding. The alteration is designated here with a switch-point and an associated fixed-segment. The later is there to ensure a valid path formation, by introducing a unit displacement from the previous line of reference. The overhead of this extension is an extra bit per controllable segment. The non-monotonic extensions does require special handling and does complicate further the decoder logic. Section 5.6 describes an alternative approach for covering non-monotonic paths, albeit using a different encoding scheme.

5.5 Practical example with s-boxes

In a typical multi-hop direct network, each node is a form of fully connected or partially connected switch-box. The example in figure 5.5 uses a fully connected switch-box. A partially connected switch-box introduces extra routing constraints that could be used to further reduce the information required to express a path.

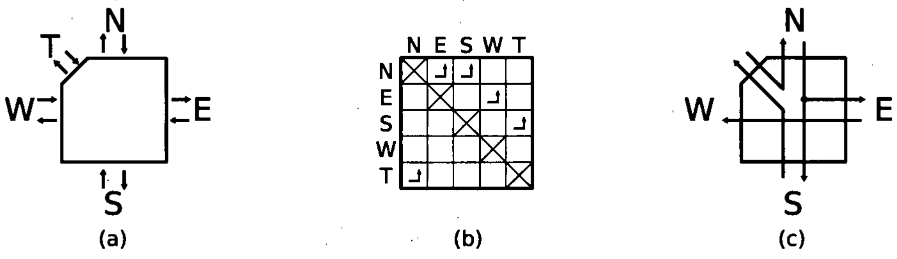


Figure 5.5: Example intra-connectivity of a 5-way sbox

The example switch-box has 4 pairs of unidirectional network ports, designated as North, East, South and West and one pair of unidirectional Terminal ports, designates as N, E, S, W and T respectively. The connection matrix of this example is shown in figure 5.5b. The arrows represent the intra-connection between an input port and an output port of the switch-box. Fort instance, an arrow on the T-N cell, defines an intra-connection between input T and output N. Figure 5.5c shows the resulting intra-connections when the 5.5b connection matrix is applied.

A switch-box may have more than one terminals attached to it, though for simplicity, the example here assumes only one terminal per switch-box. Similarly a multi-dimensional network will have switch-boxes with more than 4 network ports. Each output can be driven by any of the switch-boxs inputs. Of course an output can only be driven by one input at a time. One input can drive more than one outputs, as in the case of the North input port and South and East output ports. Figure 5.6 shows an example path and how that is mapped on a 2D array of switch-boxes.

The above decoding procedure can be performed in a dedicated unit before

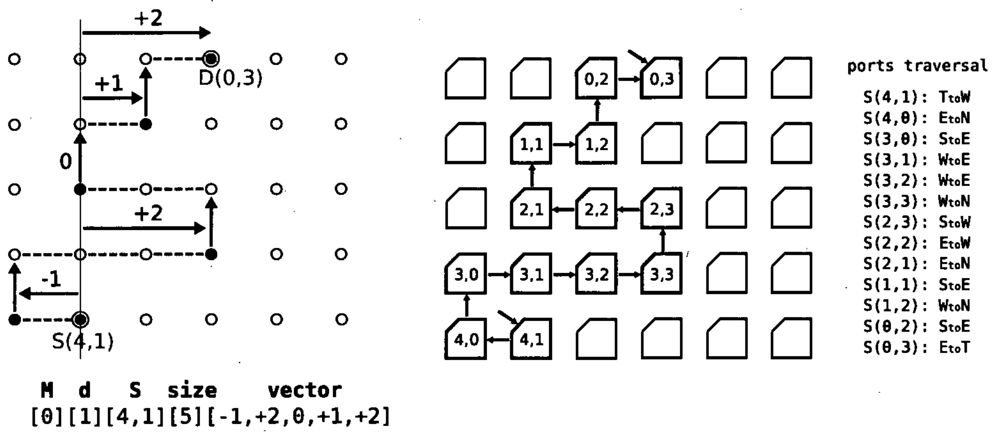


Figure 5.6: Path formation example

loading the result on the reconfigurable fabric (*off-line* routing) or on-the-fly using a set of distributed decoders, which accept the path-packet directly, to perform a collective path rendering (*on-line* routing). An off-line routing is simpler to devise and usually result in a more compact implementation. However, an on-line routing can be much faster, assuming the appropriate bandwidth is available. An off-line routing implementation is presented in section 6.1.1 and an on-line implementation in section 6.1.2 which also forms the basis of the routing mechanism used by the RICA prototype.

5.6 Turn-based format and chain encoding

The displacement-based path-encoding format ensure the formation of valid paths for any combination of its fields, This is a very useful feature to construct meta-heuristic routing algorithms. Even though for a given source-destination distance the field size is always the same, in a given set of connections with various connection distances, we end up with a variety of field sizes. Decoding those varying field sizes in hardware can be complex. A compromise is to choose the number of supported segments and have a fixed field size, a solution that trades overall bit efficiency for a simpler hardware implementation.

A more simplified derivative format is that of the *turn-based* one, which describes one *hop* at a time. It uses three turn symbols: right-turn (+), left-turn (-) and no-turn (0). With those symbols, the turn-based format can describe the *shape* of a path and is not limited to monotonic paths. However, it does not ensure the valid formation of paths for every symbol combinations, which is a unique feature of the displacement-based format. Furthermore, on average, it requires more information to describe a path compared to the displacement-based format. Nonetheless, due to the reduced amount of symbols it becomes a good candidate for a dictionary-like based encoding.

Figure 5.7 shows an example path encoded using the turn-based format. The uni-directional connection from O (output) to I (input) is meant to be rendered here in reverse order, starting from the I node. Thus, if on-line routing is assumed, the I node is the *insertion point* and thus, contrary to the natural flow, I is also the *start node.* This is done for practical reasons: an input can be driven by one output only, while an output can drive several inputs. Choosing inputs as insertion points for on-line routing, increases parallelism. Otherwise, we would need more than one injection per insertion point.

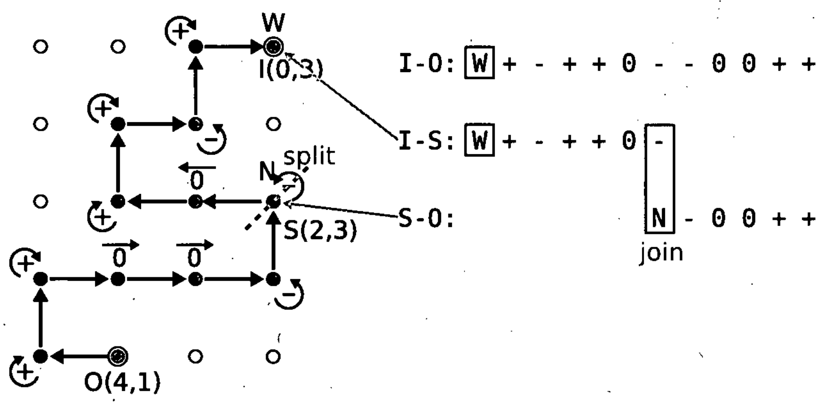


Figure 5.7: Turn-based format

The main body of the path, which is referred to here as *path shape*, is described as a sequence of turn symbols, while the direction of the starting edge is expressed explicitly. In a 2D-mesh, with the same path-shape, one can describe four different paths, each having a different end node by changing the starting-edge direction. For example, in Figure 5.7, this would involve rotating the shown path by 360 degrees in four 90 degree steps about the start node, I. In a given set of network connections it is possible to find several paths with the same shape along their full length or with the same shape along part of their length. On that simple observation, a long list of turn symbols based on a set of paths is formed, with the list of turn symbols normally defining many paths having the same shape. Using statistical analysis over a wide range of path-shapes, an efficient list or *chain* can be constructed with a high percentage of coverage, by exploiting the symbol pattern reuse that follows from the presence of the multiple same shape paths. The chain-generator tool described in section 7.6.4, is part the backend RICA toolflow, does such an analysis to generate a chain with high coverage. Using a chain of turn symbols, the shape of a path can now be expressed as a starting chain *entry* and a *length* or symbol-count. Figure 5.8 shows such an example.

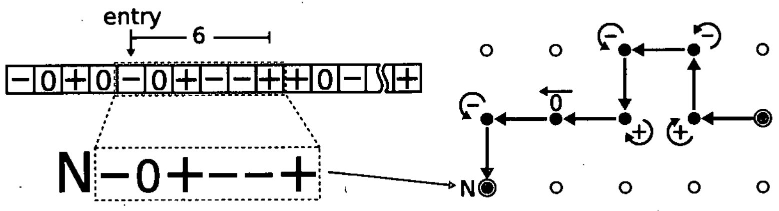


Figure 5.8: Chain encoded path

To get the whole path, we also need the direction of the initial edge. By choosing a maximum supported path-shape length, the turn-based format with chain encoding, has fixed size fields. The chain can be stored as a look-up-table (LUT) on the distributed decoding units of an on-line routing scheme. Figure 5.9 shows an example path-packet formated using chain encoding, assuming a 256 symbol chain and a maximum supported path shape length per packet of 8 symbols. Thus, 8bits are reserved for the chain entry, 2bits for the four possible initial directions for north, east, south and west and 3bits for the path length. This does not include the addressing of the insertion point, which is dealt separately by any of the schemes described in sections 5.11 and 5.12 or explicit source coordinates.

Experimental results presented in section 7.6.4, show that, out of thousands

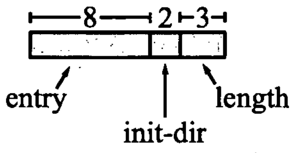


Figure 5.9: Chain encoded path packet

of connections, there are very few unique path-shapes and a list of 256 symbols is adequate to achieve close to a 100% coverage. As described in section 5.7, paths that do not match in the chain, can be split into fragments that do match.

5.7 Splitting paths and fan-outs

In case a path-shape is not covered by the chain, or its length exceeds the supported maximum length, a mechanism is required to express the path as a concatenation of smaller sub-paths. Figure 5.7 shows such a *split* point and how these sub-paths look like. The path is then rendered using two or more insertion points. Every split point injection re-establishes the last edge from the previous sub-path, in order to ensure continuity. Path discontinuity also appears in case of fan-outs. Semantically, fan-outs and split points are the same, but there is a practical difference.

Figure 5.10 shows an example set of 3 paths with 2 fan-out points, F_0 and F_1. Similar to a split point, a fan-out point divides the path into two fragments, which are rendered by injections on separate insertion points. However, to understand the difference between fan-out and split point injections, one needs to look at a normal path injection.

As described in section 5.6, the insertion points associate to input terminal ports. A path-packet describes the connection from an input terminal point, such as I_0 on figure 5.10 to an output terminal port, such as O on the same figure. The path-packet is thus injected to the sbox of input terminal port. The first edge is expressed explicitly by the 'initial-direction' field. This establishes the terminal input multiplexor's configuration. The rest of the path is rendered

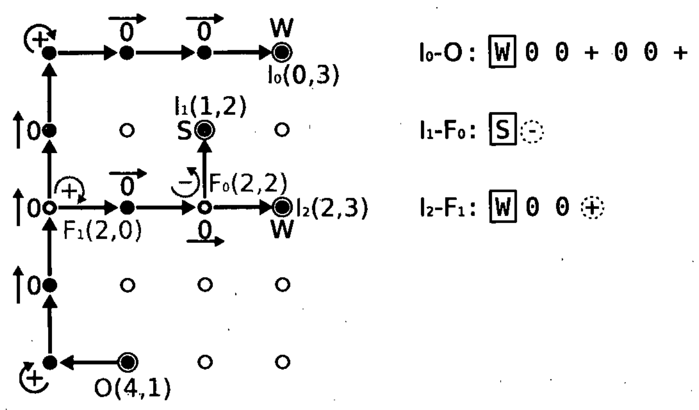


Figure 5.10: Fan-out points

based on the turn-symbols read from the chain, as shown in figure 6.3 on section 6.1.2. When the path-shape reaches its end, the rendering process finalises the path by connecting, implicitly, the output terminal port of the last sbox visited, to the rest of the path. Thus, a normal path-packet injection makes two assumptions: a path starts with an input terminal port and ends with the an output terminal port. These two assumption are not true when dealing with split and fan-out points. A split point starts from a network port, while a fan-out ends on a network port. To summarise the 3 cases:

Normal injection The path starts with an input terminal point and ends with an output terminal port.

Split-point injection The path starts with a network port and ends with a network port or an output terminal port.

Fan-out injection The path starts with an input terminal point and ends with a network port.

Based on the above, looking back at figure 5.7, the *I-S* is in fact a fan-out injection, while the *S-O* a split injection. In other words, a split injection switches the starting port to a network one, while a fan-out injection switches the end port to a network one. Thus, a path-fragment that starts and ends with a network port is a split + fan-out injection. Fan-outs are fairly common. Thus the path-packet is amended with an extra flag to indicate if the injection is a fan-out one, as shown in figure 5.11. On the other hand, split path-packets are rare and are thus handled separately, as shown in section 5.13.

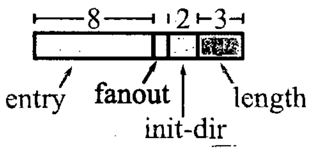


Figure 5.11: Chain encoded path packet with fan-out field

5.8 Implicit port activation

A useful side-effect of the path-encoding is the implicit flagging of a port being active in the current configuration context. Some cells have 'optional' input values, like the 'offset' of the memory interface cells described in section 3.5. In such cases, it is required to know if the port holds a valid value. Even more important than this is the required extra state each output port of an sbox needs to prevent combinatorial loops from forming.

Without this extra state, an sbox's output port that isn't configured will still route through one of it's inputs. It's thus possible to form arbitrary paths, even combinatorial loops that are undesirable. The extra state, ties the output to zero, thus preventing the propagation of signals.

All these conditions can be stored explicitly in the configuration context with 1 bit per port. However, this results in a very large increase in the configuration size. The path-encoding offers a 'cheaper' solution. Part of the decoding process, as the paths are rendered on the array, each sbox port, as well as cell port, gets implicitly flagged as active. What would otherwise be stored explicitly in the stream is now derived from the path-decoding sequence.

5.9 Clustered chain

Using a 256 symbol chain, the index needs to be 8bits. However, the entire range might not be needed on every set of connections. In a normal environment of reconfigurable fabric, there is a small set of paths that are active at any given time. This set of paths forms the active context. Within that context we might only require a small region of the chain. Again, with careful selection, a set of segments can be formed that provides enough coverage for any given active context. Doing so allows us to define a global segment for the context and have each path only define an offset within that context. Figure 5.12 shows a 256 symbol chain, divided into 4 segments of 64 symbols each. This results in using 6bits for each path, while having 2bits to define the segment for the entire set of paths of an active context. The end result is a significant increase of the bit efficiency.

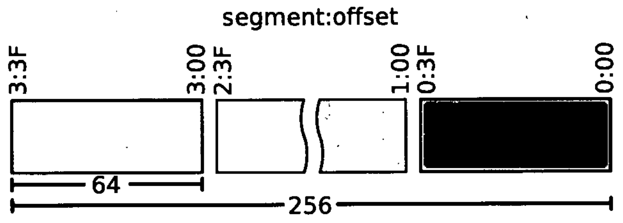


Figure 5.12: Clustered chain

5.10 Hierarchical Enable Matrix

The source field of the path packet shown in pattern 5.1, requires $log_2(X) + log_2(Y)$ bits per connection, which is 6-bits for an 8x8 array. For 11 connections the source fields will occupy 66-bits in total for a 64-element array. For such active configurations it might be better to use a 64-bit *enable-matrix*, where every element of the matrix corresponds to one insertion point and specifies if it is active or not. The above statement is summarised in table 5.1.

A scheme that performs well on both ranges is that of a *hierarchical* enable-matrix (HEM). The insertion points, or nodes, are grouped together into clus-

active connections	**suggested method**
≤ 10	source-fields
> 10	enable-matrix

Table 5.1: Source node addressing methods

ters and thus addressing those nodes becomes a hierarchical system, with domains and sub-domains, or *levels* as they are referred to here.

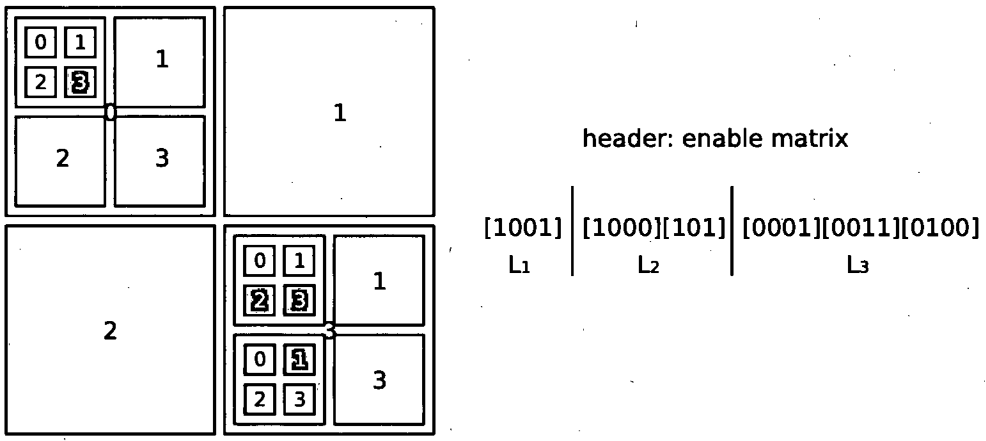


Figure 5.13: Hierarchical Enable Matrix

The hierarchical symmetric clustering of the 8x8 lattice shown in figure 5.13 has 4 elements per cluster. The leaf sub-clusters are also the nodes of the lattice. There are in total 64 nodes for this example that would otherwise require 64-bits to specify which node is active and which is not. Using the above clustering, each level of hierarchy requires 4bits as shown in figure 5.14.

The enable matrix header is thus formed by ordering these fields in sequence as shown above. Based on the motivational example of an 8x8 lattice, the maximum header size will be 4+16+64 = 84-bits. However the header size is variable and in practice it is possible to reduce the total number of bits required to represent the active clusters. This is done by ignoring sub-clusters that are not active. In the example on figure 5.13, the hierarchical enable-matrix requires only 24-bits. A typical 20% of array activity requires a hierarchical

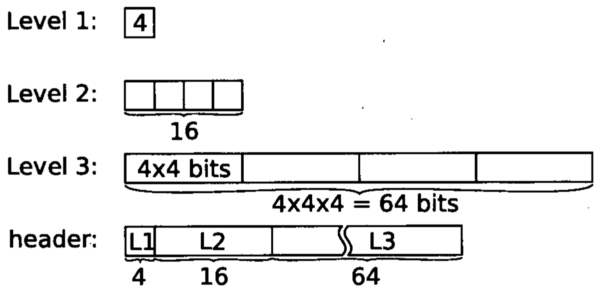


Figure 5.14: Header decomposition in a 3-level hierarchical enable matrix

enable-matrix of 48-bits on average.

The example header on figure 5.13 is built as follows. Out of the 4 level-1 clusters only two of them have at least one active node. These are cluster0 and cluster3. To represent this as a bit sequence, the above example uses a little-endian representation where the 0th cluster is represented by the most significant bit (MSB), while the 3rd cluster is represented by the least significant bit (LSB). Thus for the above example the level-1 field is [1001]. Since cluster1 and cluster2 have no active nodes they can simply be ignored. This is why level-2 field in figure 5.13 has only two sub-fields, the first one for cluster0 and the second one for cluster3. These sub-fields follow the same concept as the level-1 fields and the same applies to the level-3 fields. Note that each bit of level-3 corresponds to one node. The ones that are assigned with 1 are considered active and thus a path-encoding field (payload) should be present in the stream. The order in which the level fields appear in the header also dictate the order in which the path-encoding fields should be placed in the bit-stream, to maintain the correct association.

5.11 Hierarchical clustering for loading

One of the big concerns in flexible interconnection structures is the ability to propagate the configuration information across the lattice. This poses a significant implementation issue and often forms the bottleneck of the system, due to high bandwidth requirements. In the path-encoding technique, each point-to-point connection is described as a set of path directives starting from the source to the destination node. The encoded path is a compact information packet (unit) that can be loaded into the source node. Path formation/rendering can be done on-the-fly in a wormhole-like fashion, in an asynchronous or synchronous manner, as described in section 6.1.2.

On a typical configuration context, the number of insertion point nodes is only a fraction of the total number of active nodes. Because of that, the required number of parallel insertion points can be reduced significantly. This opens the possibility for a less interconnected loading network, with smaller bandwidth requirements. An example scenario with a quarter of the bandwidth requirements of the fully connected loading network is shown in Figure 5.15.

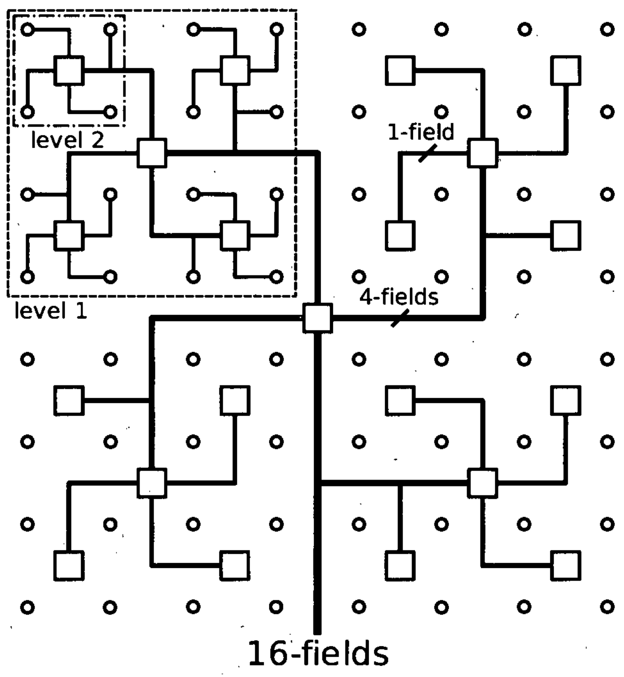


Figure 5.15: Hierarchical symmetric loading network

For a 64-element array the fully connected loading network requires 64 parallel fields. This could allow all nodes to be loaded simultaneously. The above example hierarchically clustered network has only 16 parallel fields which are evenly distributed across the array. This allows 16 simultaneous fields to be loaded to each of the 16 level-2 clusters. Each level-2 cluster encapsulates 4 nodes. In general, to load all the nodes of a level-2 cluster it requires 4 cycles. This level of parallelism is sufficient for typical network utilisation; that is, the number of active point-to-point connections.

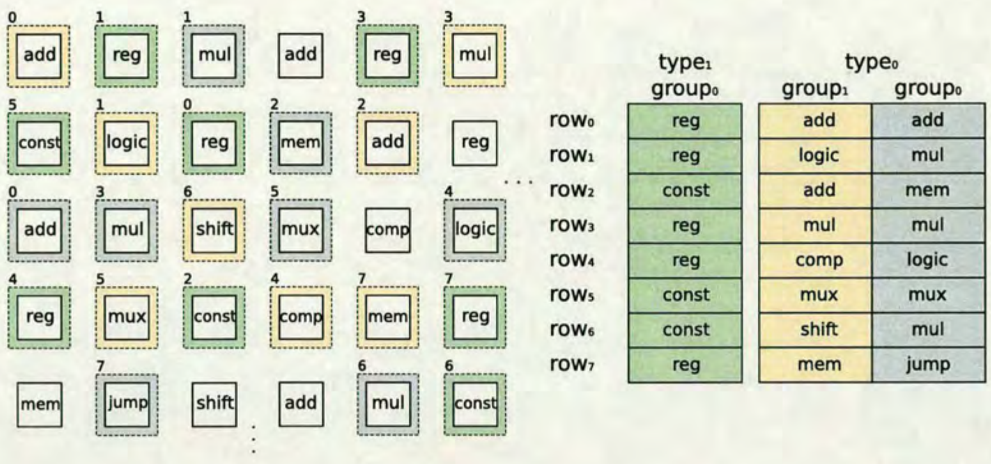
5.12 Irregular/Asymmetric grouping

Using clusters, the numbers of cycles required to load a configuration depends on the number of active insertion points and where these are located, as seen above. To reduce the average number of cycles required, an analysis can be applied to determine which insertion points to group together to form a cluster. A tool that does such an analysis is described in section 7.6.3. This will provide clusters that may not be spatially contained in one region, but rather could be placed physically anywhere in the array in an irregular, asymmetric fashion. As such, this approach represents a generalisation of the regular, symmetric approach described above with reference to figure 5.15. The irregular disposition of clusters of the present approach may introduce some extra complexity on the physical wiring to the clusters, but it improves the average loading time for a given placement. One thing to note is that the hierarchical clustering described in 5.10 can lend itself to an non-spatially contained physical cluster formation, such the one shown in figure 5.16. This figure shows the separation of insertion points into types, which is an implementation detail described in section 5.13.

The hierarchical enable matrix can be disassociated from the physical positions of the insertion points it refers to. Doing so improves the efficiency of the HEM encoding.

5.13 Context partitioning. Rows and groups

The HEM encoding results in a variable configuration word. If all the elements of a top level cluster are inactive, the cluster is omitted from the stream. This variability offers good bit efficiency, but can result in a more complex decoding



	type₁	type₀	
	group₀	group₁	group₀
row₀	reg	add	add
row₁	reg	logic	mul
row₂	const	add	mem
row₃	reg	mul	mul
row₄	reg	comp	logic
row₅	const	mux	mux
row₆	const	shift	mul
row₇	reg	mem	jump

Figure 5.16: Groups

process. The following describes a way to eliminate the variability seen in 5.10. It's important to note that this method trades bit efficiency for implementation simplicity.

The configuration context is split into *rows*, which are populated to differing extents, depending on the insertion points that are used, as shown on figure 5.17. Each row contains one member slot (field) for every cluster, which may be populated or not. The configuration context has as many rows as the element count of the most populated cluster. Each row matches and drives the configuration distribution network, which can be something like the one seen on figure 5.15.

Figure 5.17 shows the configuration context partitioning used on the RICA prototype chip. As in described section 5.6, an insertion point corresponds to an input port of a connection pair. Section 3.1 shows that the RICA cells have one, two or three inputs. Looking at Appendix C, most cell types have 2 inputs ports. When a cell is used, it's most likely that all of its input and output ports are in used. This is definitely the case for combinatorial cells, such as adders and shifters. In an attempt to reduce the number of insertion points, those that are part of the same cell are grouped and delivered together. Fewer insertion points means less IP address fields, which would otherwise be

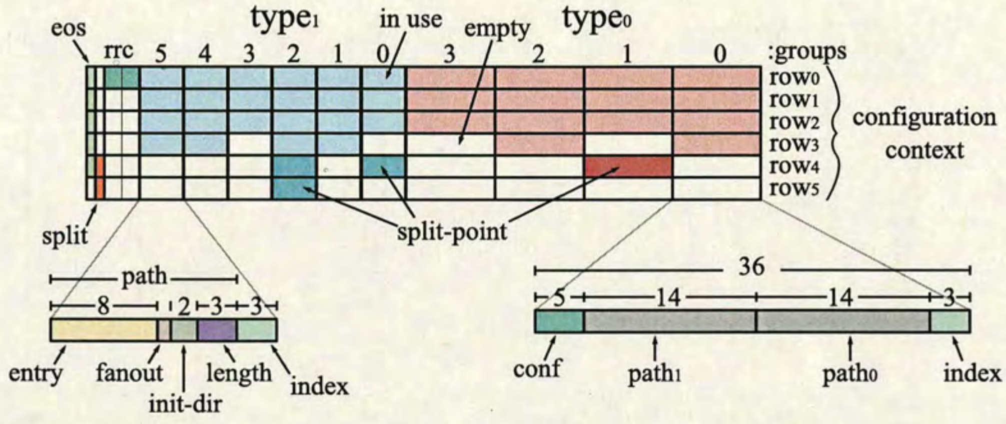


Figure 5.17: Configuration context rows

redundant. The aggregated insertion points are referred to here as *type*$_0$, while the single ones as *type*$_1$. Figure 5.16 shows an example grouping of a section of the core and the two insertion point types. One can also see the cell types that occupy each insertion point type.

Figure 5.17 shows the path-packets, which have now been amended with the insertion point index (IP address) within the given cluster. As discussed in section 5.7, the split-path are handled separately. To avoid adding an extra bit to every packet, the split paths are placed at the end of the configuration stream and a per row flag indicates that all the path-packets in that row are split ones. Although it seems a waste of space, split paths are very rare and thus the overall impact is smaller than what it would have been otherwise if an extra bit was added to each packet.

Other notable features on figure 5.17 include the *eos* field, which stands for 'end of step'. This flag indicates the last row of the current configuration context. The *rrc* field includes timer fields for the read memory interfaces and the step time, controlling how long the configuration context should be retained before switching. This is described in more details in chapter 6, section 6.2.

5.14 Compressibility

Each of the outputs of the example switch-box requires at least 2-bits to distinguish between the 4 possible intra-connections. In practice an extra output state for the un-driven case is required, to prevent combinatorial loops, which normally ties the output to a logical zero. Normally, this would require to be stored in the bitstream. However, as explained in 5.8 the path decoding process will set or un-set those control bits implicitly, thus eliminating the need to store them in the stream. Although important, this extra state is ignored on the following analysis, thus keeping the bits 2 per port.

For a total of 5 output ports, 4 network and 1 terminal one, each switch-box requires a configuration of 10bits in total. The entire array would require N $\times$ 10bits for the interconnection alone, where N is the number of sboxes. For a 64 sbox array, that is a total of 640bits for interconnect configuration. Even if there is only one active connection, which occupies a small subset of the network resources, we still need to provide the total amount of configuration bits including those that are not used. Clearly there is a significant amount of redundancy in the configuration stream, which can be removed. An encoding technique that stores only the active switch-boxes would require for the example of figure 5.5 at least $13 \times 10 = 130$bits in total, since the example path occupies 13 switch-boxes. This represents the raw useful information required for this example. In practice more than that is required since some extra information is need to be able to re-assign the appropriate bits to the correct switch-box.

Based on 5.1 and assuming a 8×8=64 switch-box lattice, the path-encoding technique requires $log_2(8) = 3$bits per axis for the source field, 2bits for monotonic and direction flags, 3bits for the size fields, assuming a maximum displacement vector size of 8 and another 3bits per controllable segment (5 of them). That is:

[m]	[dir]	$[S(y_s, x_s)]$	[size]	$[d_4 \quad d_3 \quad d_2 \quad d_1 \quad d_0]$	
1	1	6	3	12	= 23-bits

Compared to the minimum configuration required for the participating switch-boxes of the example path, the path-encoding technique achieves a $20/130 = 0.18$ compression ratio. Worth mentioning again, is that the uncompressed configuration size for the above scenario is 640bits no matter how simple or complex the active connections are. Thus the actual compression ratio achieved is $23/640 = \mathbf{0.035}$.

In practice, the compression ratio varies depending on the core utilisation. Table 5.2 shows various examples encoded with the path-encoding technique and the average compression ratio achieved. The lower the compression ratio is, the better. Each of those examples is composed of a set of configuration contexts, with varying core utilisation as seen by the standard deviation enclosed in parentheses.

	average utilisation	raw	PE	compression
libmad	12%(6.43)	1,507Kb	329.64Kb	0.22
DCT	18.54%(9.85)	44.7Kb	15Kb	0.34
h264	10%(5.32)	23,221Kb	4,644Kb	0.2

Table 5.2: Path-encoding compression examples

Figure 5.18 shows the relationship between core utilisation and compression ratio achieved in the MP3-libmad example, mapped on a 9x9 core. This off-the-self MP3 implementation has more that 1.2K contexts and achieves a core utilisation that ranges from 2.67% up to a close 50%, with compression ratios ranging from 0.1 to 0.75. It thus offers a good set of data to analyse the path-encoding technique.

The first observation is that the compressibility reduces as the core utilisation increases. In fact, following the extrapolated line, the path-encoding is not able to provide any benefits for core utilisation near 50%. Anything above that and the encoding expands the bit-stream instead of compressing it. This is a common characteristic of any code compression technique. On a practical implementation it would be best to disable the path-encoding, bypass the decoder and load the raw stream directly instead.

Other than the total core utilisation, the compression ratio depends on the contents of the configuration context as well. This is shown by the range of

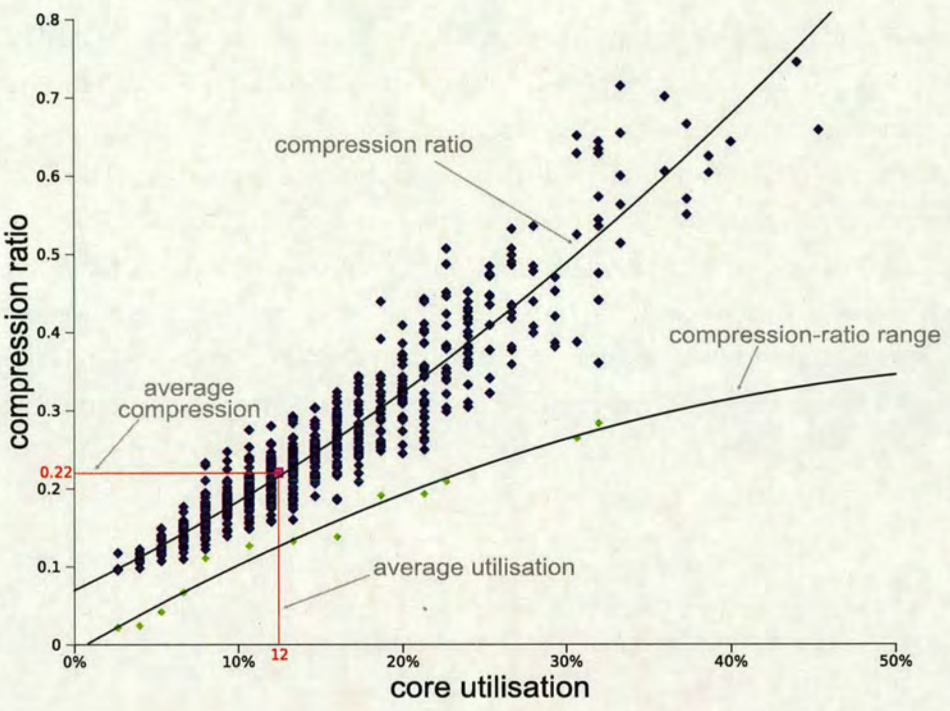


Figure 5.18: MP3-libmad compressibility

compression ratio in the graph for different core utilisation values. The second set of data on the bottom part of the graph, show how this range changes in relation to the total core utilisation. The extrapolated line shows that for high utilisation the compression ration range starts to saturate, which is only natural, since as the total core utilisation increases there are fewer possible permutations.

The purpose of code compression, such as the path-encoding technique, is to reduce the footprint of under utilised configuration contexts, what in VLIW terminology would ammount to No Operation (NOP) slots. The goal of compiler optimisation is to increase overal core utilisation and thus reduce those NOP slots. However, due to the very nature of most applications, eliminating those NOPs is not possible. The path-encoding offers a solution here, occupying only 12% of the total design area, as seen in chapter 3.

The effectiveness of the path encoding as a compression scheme can be seen by looking at the program memory bandwidth requirements of an application, such as the MP3 decoder. Figure 5.19 shows the accumulative configuration bits transferred from the program memory over the process of decoding 5 seconds worth of MP3 audio data, with and without path-encoding. The graph is composed by grouping the various executed configuration context based on their core utilisation figures. Therefore, for every core utilisation figure, the graph shows the context with the highest bit transfers. The transfer size depends on the number of times each configuration context was fetched as well as the size of the context itself. What the data shows is that, despite the diminishing returns in code-compression as the core utilisation increases, there is a total of **2.75** times reduction in total program data transfers.

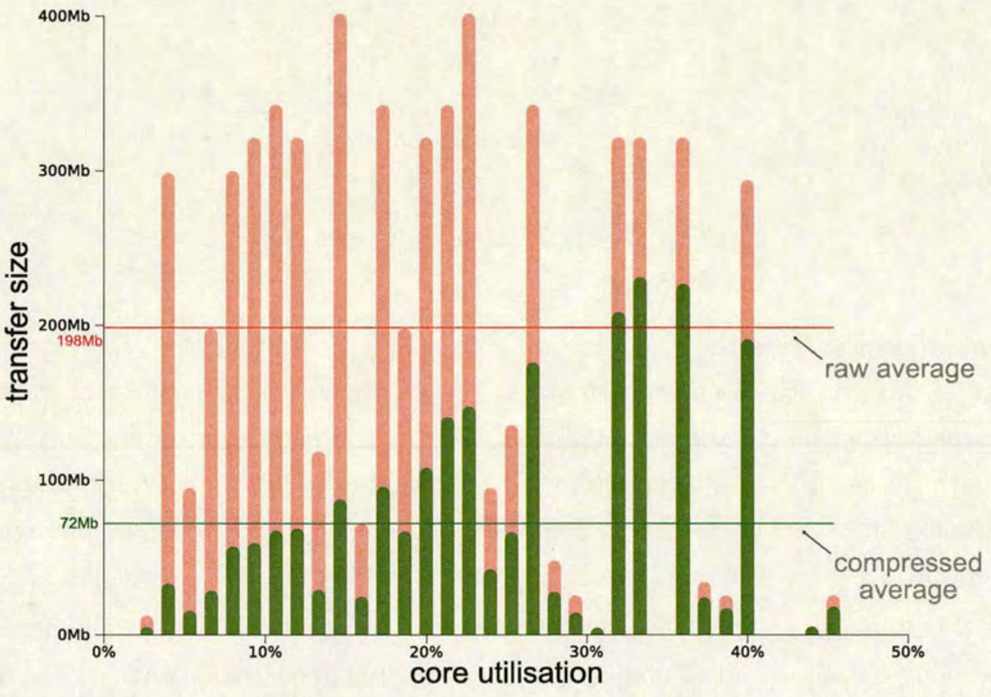


Figure 5.19: MP3-libmad configuration transfers per core utilisation

Chapter 6

Reconfiguration

6.1 Configuration distribution

The configuration distribution network is responsible for delivering the configuration stream to the various parts of the reconfigurable fabric, from the program memory store. The simplest form of this network is a set of point-to-point wires driven by the program store itself. The configuration stream is stored in raw format in that case. A reasonably sized core of a few hundred computational units, has a fairly wide configuration context, in the range of a few kilobits. In case of an off-chip program store, it would be impractical to have such a wide distribution network. Thus the configuration stream is normally spilt into smaller, more manageable chunks, which are delivered in sequence. The core assembles those chunks into a local store for the entire configuration context. The same argument applies for on-chip program store of several hundred contexts. A kilobits wide memory would require a custom design with serious design challenges.

The benefits of code compression are addressed in 5.14, which shows the total bandwidth savings achieved by applying compression to low utilised contexts. When code compression is applied, the configuration stream needs to be decoded before it is mapped to the core. There are two main options for the configuration distribution network in this case. It can either deliver the encoded stream to the core, where the decoding happens, or can come after the decoding process and thus deliver the raw stream instead. The first scenario is referred to here as *on-line* routing, while the second *off-line* routing.

6.1.1 Off-line routing

The decoding process is normally an iterative one, which, in the case of the path-encoding, involves the unfolding of each path based on the encoded information in the configuration stream. In the case of an off-line routing, other than the decoding processes, for the re-configuration to be complete, the resulting configuration patterns need to be applied to the right routing resource to establish the paths and setup the operation modes of the right cells.

There are two main ways to implement the configuration loading, after decoding. It can either be done as a memory mapped storage space or loaded in a stream-like fashion. In the memory mapped scenario, the configuration context is accessible as a random access memory (RAM). The decoding process can update any section of the configuration context, thus strict order is not required. On the other hand, a streaming-like loading process requires the input stream to be the right order. Depending on the implementation, the streaming approach may require a fixed number of cycles to load a configuration stream. Due to the nature of the path-encoding, which follows the shape of individual tracks, it is difficult to devise a streaming-like loading system without employing 'on-line' routing practises. Because of that, only the memory-mapped approach is covered here.

$$[\mathtt{m}] \quad [\mathtt{dir}] \quad [\mathtt{size}] \quad [\mathtt{d}_0 \ldots \mathtt{d}_{n-1}] \tag{6.1}$$

The following assumes an enable matrix is used for addressing the insertion points, as seen in 5.10, and the packets are of the form found in 6.1. The HEM needs to be decoded first, before being able to decode the path packets. However, decoding of the path packets can start once the fields that correspond to the current subset of packets being decoded is ready. Due to the hierarchical structure of the matrix, it is possible to do an incremental decoding, alongside the path packet decoding process. A level-2 cluster of insertion point indexes can thus be derived using the 3 fields corresponding to that cluster, as seen in Figure 6.1.

Every HEM field encodes n indexes, where n is number of sub-clusters and thus bitwidth of the field. In general, the index of an insertion point is the binary concatenation of the decoded indexes the corresponding set of HEM fields from each level, $index = \{l_1, l_2, l_3\}$. Based on this inherent property of the HEM, it is possible to interleave the HEM fields with the path-packets in the stream, as shown in 6.2. This allows a more uniform memory access and an incremental decoding process.

$$[\mathrm{L}_1]\left[[\mathrm{L}_2]\left[[\mathrm{L}_3][\mathrm{PL}_0]\cdots[\mathrm{PL}_n]\right]\cdots\left[[\mathrm{L}_3][\mathrm{PL}_0]\cdots[\mathrm{PL}_n]\right]\right]\cdots\left[[\mathrm{L}_2]\left[[\mathrm{L}_3]\cdots\right]\right. \tag{6.2}$$

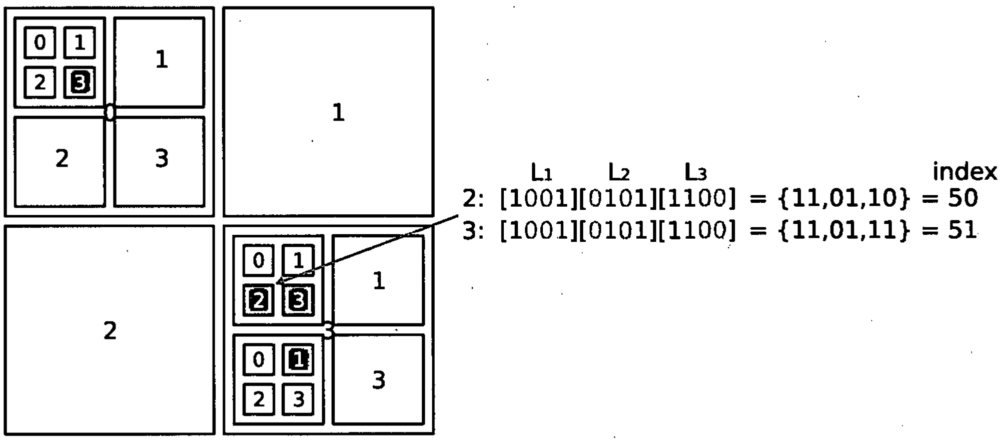


Figure 6.1: Incremental HEM decoding

$\boldsymbol{L_n}$ are the HEM fields.

$\boldsymbol{PL_n}$ are the path packets.

Due to the variable nature of both the HEM and the path-encoding, the decoder consumes a variable number of bits at a time. A word assembly module plays the role of an interface between the decoder and the program memory store, fetching the right amount of data from the memory and delivering properly aligned fields to the decoder. The general overview of the decoder system is shown in Figure 6.2.

The 'memory interface' module deals with the variable size requests and is what does the word assembly. The decoder does an incremental HEM decoding, by concatenating L_1, L_2 and L_3 fields as shown in figure 6.1. The path-packet decoding is state machines. The 'GET HEADER' stage collects and sets the monotonicity and direction on the monotonic axis of the path as well as the size of the displacement vector. The rendering process iterates through the displacement fields and derives the edges and vertices of the path based on those.

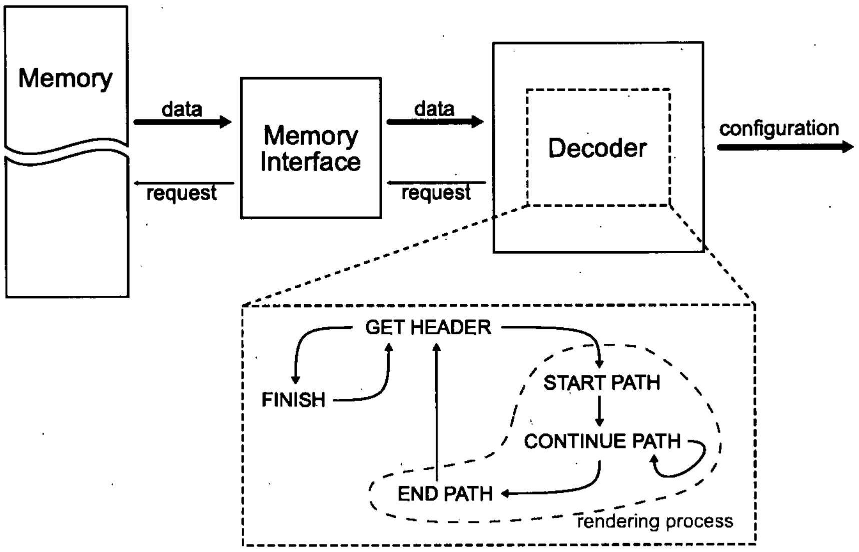


Figure 6.2: Off-line HEM and path decoder

6.1.2 On-line routing

Considering the routing configuration needs to be propagated across the routing network in order to render the tracks of the current context, an alternative solution would be to decode the path 'in place'. Instead of propagating the decoded stream, the fetching unit can propagate the relevant fragments of the encoded stream to the right place, inside the routing network, to be decoded. This is the basic concept behind the *on-line routing* approach, which resembles the pipelined propagation of a wormhole routing.

There are several ways one could partition the encoded stream into autonomous fragments. Treating each point-to-point connection as one fragment is probably the smallest granularity, in which case, the fragment would be a path-encoding packet. Each point-to-point connection forms a *track* inside the routing network, from the source node to the destination one. The shape of this track is described in the relevant path-encoding packet. The fetch-unit delivers this packet to its source node, where the decoding processes commences. The decoding process is done here in a distributed fashion, where each node includes a small decoding logic. The track 'rendering' completes when the packet reaches the destination node.

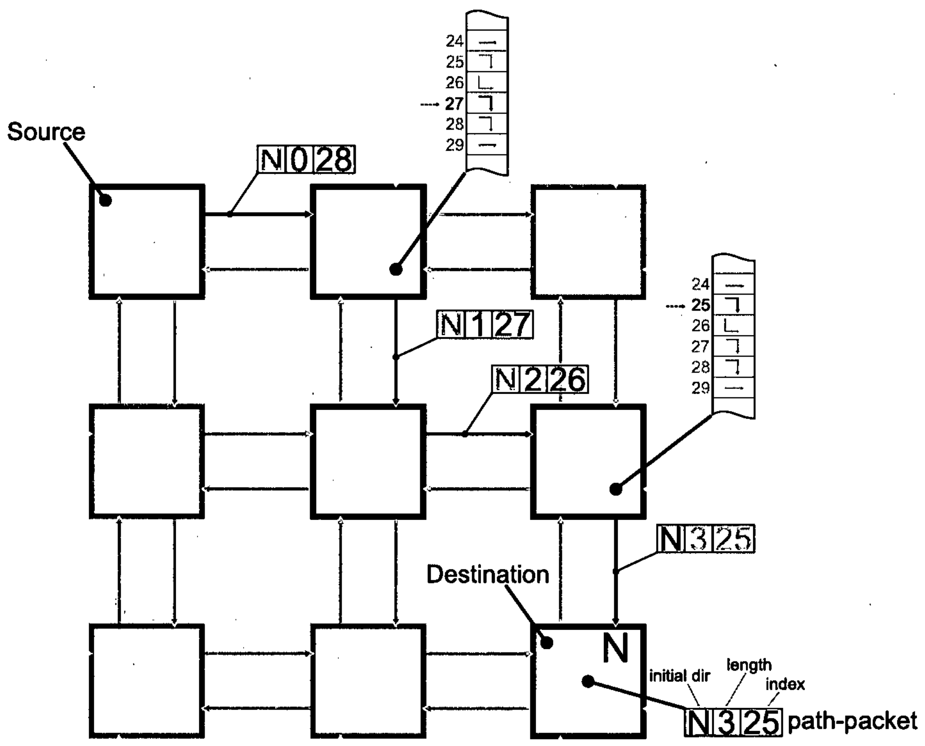


Figure 6.3: On-line path rendering

Figure 6.3 shows the process of rendering a path on-line. The packet is encoded using the turn-based format with chain-encoding presented in section 5.9 and as discussed in section 5.6, this is rendered in reverse order, from the destination port to the source port.

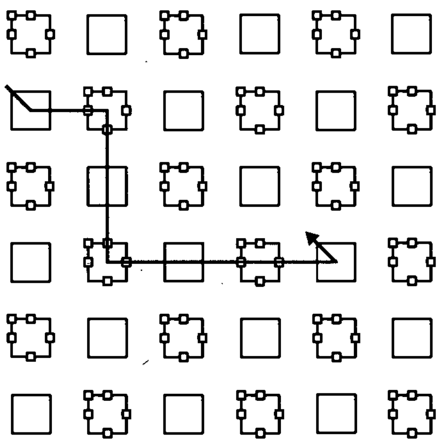
6.2 The Reconfiguration Rate Controller

The active time of a configuration context depends on the critical path of its longest data-path. Due to the highly flexible interconnects of the architecture, data-paths can be formed with arbitrary shapes and lengths. The critical path of a context can thus vary from a few non-seconds to over hundred nanoseconds, depending on the size and mixture of cells in the array. Using a reasonably sized fixed context duration would limit the longest critical paths one could form and would introduce significant idle time for short critical paths. on the other hand, choosing a short fixed duration will limit the operation chain depth one can achieve. Clearly, the architecture should cope with the variable length contexts.

Long critical paths are normally tackled with pipelining. Conventional RISC processors use pipelining to reduce the critical path of their fetch - decode - execute - memory - writeback circle. However, these often deep pipelines introduce a latency, which degrades performance when branching. To avoid the effects of this latency, modern processors employ sophisticated branch prediction units. In the arms race for faster processors, the branch prediction units are the crown jewels of each architecture and are thus close guarded industry secrets.

An important contributor to the critical paths of a reconfigurable fabric, is its interconnection network. If a fixed pipeline is to be used, the interconnects should be pipelined as well.To avoid imposing limitations to the routing flexibility of the network, the minimum permanent pipeline stage size for a 2D toroidal mesh is equal to the propagation delay of two switch boxes, as seen in figure 6.4a. A wider pipeline stage configuration introduces uneven path delays, as seen in figure 6.4b. Here the pipeline registers are inserted at the output ports of the selected switch-boxes. This requires 6 output registers (4 network and 2 terminal) per switch-box. A 64-element 2D toroidal array would thus require 192 registers to pipeline its network.

The pipeline registers have an effect on the overall silicon area, but are not the main problem of such permanently pipelined interconnects. Path balancing and synchronisation are what make such a design complicated. A software solution could involve balancing out the pipe-line depths of the various parallel data-paths, by inserting extra pipeline registers to the shortest ones. This becomes a complex routing problem, since extra registers means extra 'hops'. Hardware synchronisation techniques would involve a completion detection mechanism for every parallel data-path. These issues are tackled in asynchronous design methodologies. An asynchronous design is normally based on some kind of hand-shaking mechanism [55], thus becoming self-timed, where


(a) symetric 2 switch-box stage size

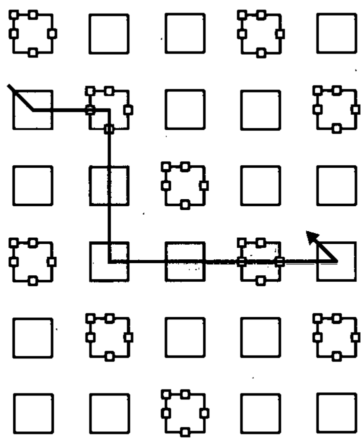

(b) asymetric 3 switch-box stage size

Figure 6.4: Pipelined interconnects

the critical path propagation delay is dynamically derived. An asynchronous design can also be seen as an 'event driven' one. Once no *events* are left to serve, the execution of the current configuration context is considered complete and the core may switch to the next one, or re-execute the current one in the case of a loop. Even though it is a very attractive solution, asynchronous designs are still in their infancy. However, they are gradually winning the trust of the semiconductor industry [56, 57].

Software pipelining, which is supported by the current RICA implementation, can reduce the critical paths and increase parallelism and thus throughput. This involves adding registers in long data-path chains, thus splitting the source configuration context into logical pipeline stages. The effectiveness of this approach depends on the number of available registers in the core. Furthermore, software pipelining using registers as cells, requires more routing resources to access those registers and has diminishing returns as core utilisation increases. Software pipelining is better suited to an architecture that incorporates controllable registers as part of the interconnects. Software pipelining is a performance enhancing features used in kernels, rather than a solution for the variable length paths. Although one can use software pipelining to enforce a fixed step clock, applying it to non kernels introduces unnecessary latencies.

The solution chosen for the RICA prototype is based on a conventional synchronous design that handles the configuration context duration variability using timers. The Reconfigurable Rate Controller (RRC), as the name implies, manages the reconfiguration of the core. The critical path of each configuration context is calculated by the compiler/scheduler and is included in the configuration stream as a timer time-out. The RRC, which in simple terms is composed of a set of timers, counts until the execution timer expires, before issuing the next context-switch. The timer resolution is defined by the master clock of the device, which is also feeds to the RRC.

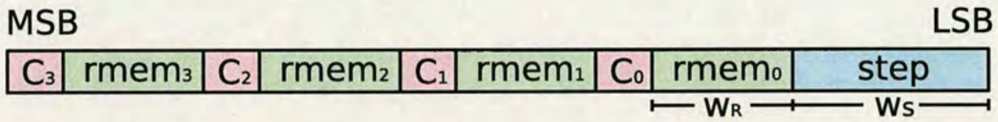


Figure 6.5: RRC fields

Figure 6.5 shows the various bit-fields of the RRC configuration, with the *step* field representing the configuration context/step execution timer. The width of the *step* field is W_S and depends on the timer resolution as well as the maximum allowed critical path. Other than the critical path duration time, the RRC's configuration also includes the starting times of the read-memory access requests of the relevant interface cells, shown here as $rmem_x$. The bit-width of those fields is W_R and depends again on the timer resolution. The C_x is the *cascading bit* and is explained later.

Unlike write-memory accesses that occur at the end of the execution of a configuration context, read-memory accesses form parts of the various data-paths. It is important to issue the read-memory access request once the address is ready. For that, we need to calculate the propagation delay of the data-path feeding the address input of the read-memory interface cell. This propagation delay forms the $rmem_x$ start-time of the read-memory interface cell, as seen in figure 6.5.

After a read-memory interface cell (*rmem* cell) issues a read request, its output will be ready after the memory latency time has elapsed. As discussed in section 3.5, simultaneous accesses to the same memory bank are queued

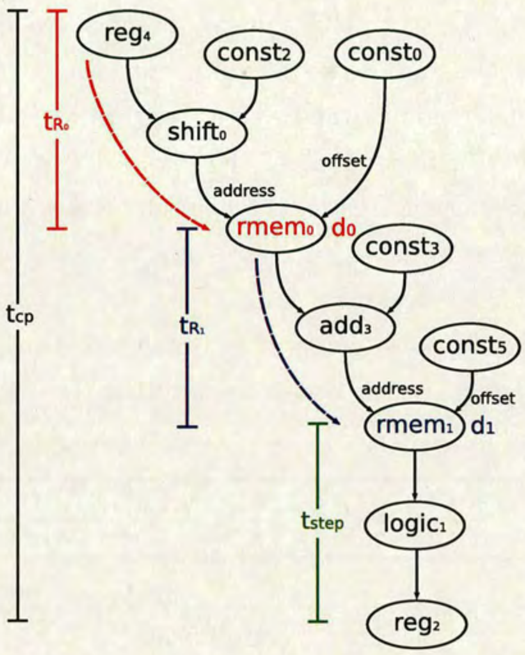


Figure 6.6: RRC read-memory interface start time

by the arbiter. The total memory latency is thus unknown and depends on run-time conditions. To overcome this problem, the arbitration logic signals the RRC when a read memory operation completes. This way, the RRC can incorporate the run-time delays in the total execution time.

Figure 6.6 shows an example critical path with two *rmem* cells. The total 'critical path' is t_{cp}. The combinatorial delay of the path feeding the *address* input port of $rmem_0$ is t_{R_0}, while the memory latency for $rmem_0$ is d_0. In this graph, $rmem_1$ cascades to $rmem_0$. Its starting time would normally be $(t_{R_0}+d_0)+t_{R_1}$. Since d_0 is not known at compile time, the two *rmem* timers are 'cascaded'. The start timer for $rmem_1$ will start once $rmem_0$ has completed its operation and its output is ready. This accounts for the $(t_{R_0}+d_0)$, leaving the t_{R_1} for the $rmem_1$ start timer. This 'cascading' condition is indicated using the cascading bit C_1. Similarly, the *step* timer will start once all *rmem* operations have finished and thus it only requires to count the t_{step} time. Overall, the total critical path is $t_{cp} = (t_{R_0} + d_0) + (t_{R_1} + d_1) + t_{step}$, where t_x are static delays, while d_x are dynamic.

6.3 Pre-fetching

The reconfiguration of reconfigurable core such as RICA can take several cycles. The configuration latency introduces can impact the overall efficiency of the core and thus reducing that latency is important. There are ways to hide some of that latency by pre-fetching the configuration into a 'back page' context storage, ready to be used when needed. There are 3 simple cases where the next context can be safely pre-fetched:

a No branching is present in the current context/step.

b The branching is unconditional and the address of the next context is provided as an immediate value.

c We are in a *kernel*. The exit step can be prefetched .

By pre-fetching the 'next' context, the configuration time can partially or completely overlap to the execution of the current step, thus reducing the effect it has to the total execution time. Figure 6.7 depicts on such pre-fetching scenario and compares it to the non-prefetching case. The *core* row shows the general switching activity on the core, with A to D corresponding to different context, the *fetch* re presends the configuration loading subsystem and the $bank_x$ shows the activity of the data memory banks. As described in section 6.2, the memory reads are issued one or more cycles after the start of the execution of a context, which is shown here by 'synch'. Memory writes are issued at the end of a context and can run in parallel to the core execution. However, they can block subsequent reads.

The top case corresponds to the non-pre-fetching scenario, while the bottom case to the pre-fetching one. When the core does not pre-fetch context, fetching always happens at the end of the execution of the current context and delays the start of the next context. However, when pre-fetching is enabled, the fetch unit can start loading the next configuration context whilst the core is running

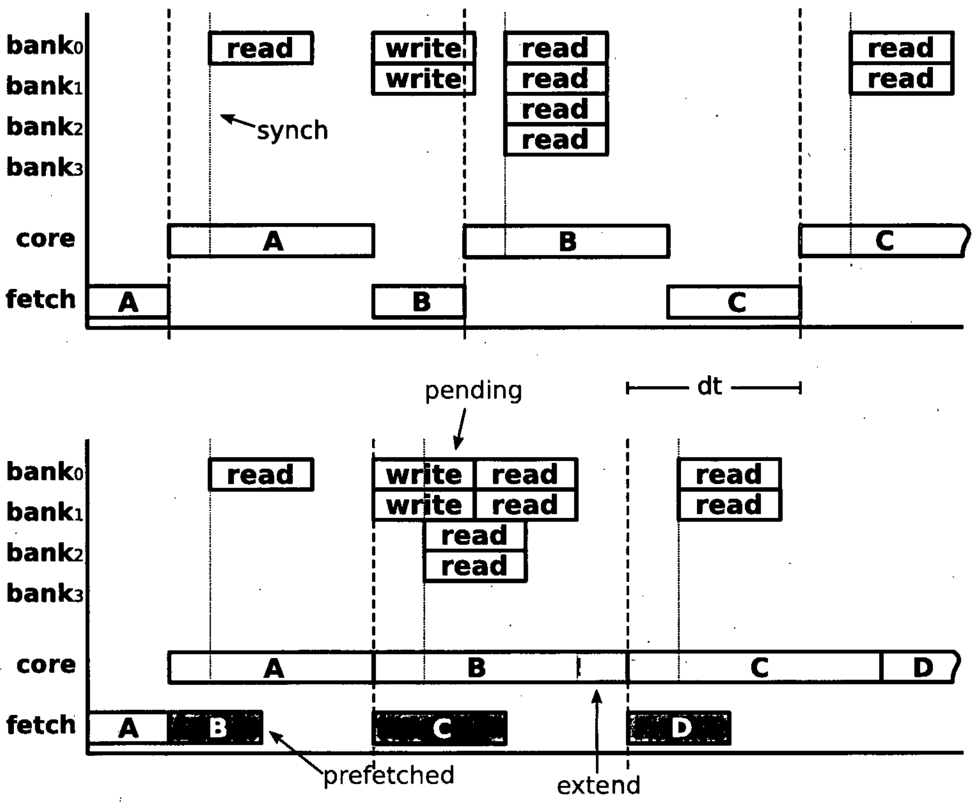


Figure 6.7: Prefetching and dynamic delays

the current one. This is show in figure 6.7 with B, C and D configuration context overlapping in time to A, B and C context respectively. This time 'compression' introduces some rather interesting effects. One such is that of memory writes blocking memory reads. Since context B starts now earlier than before, the memory writes of context A conflict now with the memory reads of context B. This causes the core to delay the retirement of the current context B, shown here as an 'extend' in the time-line. Thus, read after write cases can reduce the effectiveness of pre-fetching, although in general the improvements are significant.

The results in table 6.1 show the effect pre-fetching has on the libmad MP3 decoder running on RICA. The core is clocked at 50MHz, which includes the

RRC and the program and data memories.

	pre-fetch	non pre-fetch
avg ns/step	73.96	88.43
total time	2.5	2.99
config latency	0.5	0.99
MSamples/sec	13.52	11.31
step count	33839689	33839689
fetch count	17379147	17379147
pre-fetch count	10270603	0
pre-fetch percentage	59.1	0

Table 6.1: Results with and without pre-fetching on libmad

Table 6.1 shows a reduction of about 50% in configuration latency, which amounts to an over 16% improvement in final throughput. Figure 6.8 shows the dynamic data retrieved from the RICA simulator. The graph shows the activity on a kernel step and the exit step that follows. The highlighted bars represent the fetching loading time. The bottom case has pre-fetching enabled, while the top case does not use pre-fetching. Thus, in the upper case, the core has to wait for the configuration context to load before it can start executing. The green vertical lines on the right hand side contexts, show the start of execution for each of those cases. The time delay between the non-prefetched and pre-fetched case is marked as d_t.

The results show that, even without the use of complicated branch prediction, the simple pre-fetching offers a healthy performance boost. From an implementation point of view, pre-fetching requires an extra set of configuration context registers, which for the RICA prototype chip represents a 5% area increase on a $4mm^2$ total core area.

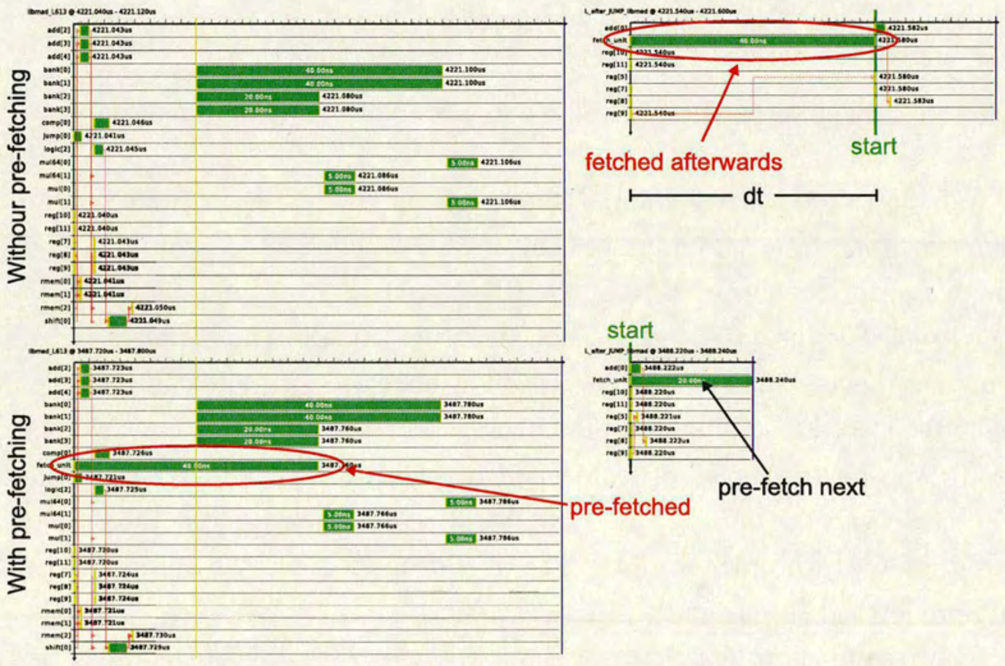


Figure 6.8: Time-charts with and without pre-fetching

Chapter 7

Software Toolflow

7.1 Toolflow overview

Several aspects of the design flow of reconfigurable architectures are similar to those of an ASIC design flow, such as place and route. On the other hand, since reconfigurable architectures are programmable in nature, parts of their design flow resemble that of GPPs, such as the compilation phase. The RICA toolflow is no exception, combining stages from both ASIC like and GPP like toolflows. Furthermore, since RICA is an architecture template, it also provides a set of tools that aid to the development of different core configurations, such as size and resource mix. The software toolflow is thus split into two sets of tools, one targeting the software developer and the other the core developer.

7.1.1 Frontend

The following list of tools are what an application developer would use when writing an application for RICA.

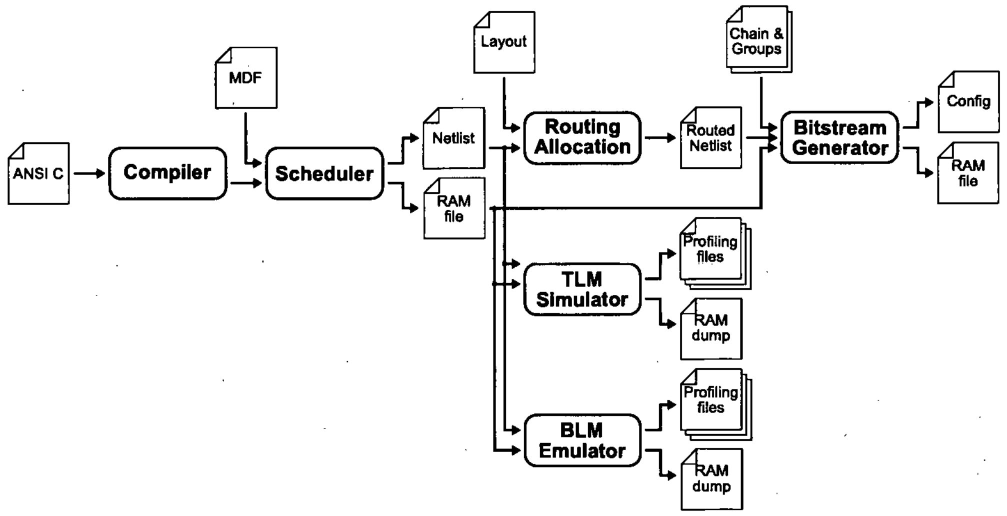


Figure 7.1: Toolflow overview: Frontend

Compiler The design entry for RICA is ANSI C, although, since the compiler is based on GNU GCC, other language front-ends could also be used[58].

A RICA specific compiler backend is provided that emits a suitable assembly format. This backend also attempts to enable particular optimisations that are better suited to a RICA design, such as inferring multiplexor logic in certain branch conditions, thus extending the use of the conditional move instruction. The modifications to the compiler are not part of the present work and thus are not covered here.

Scheduler This is a stand-alone, custom tool that accepts the output assembly produced by the compiler. It also reads in the machine description file (MDF), which gives details of the target RICA core, such as resource mix and propagation delay per cell type. The compiler does not deal with the spatial nature of the core. The scheduler re-builds the control flow and data flow graph of the application from the assembly and based on the MDF, determines appropriate mappings, which form the operations of a configuration context. The application is thus divided into a set of contexts with their associated control flow information dictating the execution flow. These contexts, which are known as 'steps' in the RICA terminology, are then output in the RICA 'netlist' file format, which is described in Appendix B. The global memory contents are also extracted from the assembly and saved in a separate file. This separation is done because RICA is a Harvard architecture with separate data and program memory. The scheduler does also perform some data-flow graph simplifications not picked up by the compiler. The implementation of the scheduler is a separate project which is not part of this work.

Routing-Allocation Performs final allocation and routing of the design. Both the compiler and scheduler operate without a physical representation of the core itself. The compiler knows only about the supported type of instructions, while the scheduler has extra information on the resource mix, but not the physical placement of the core. The routing-allocation tool, described in section 7.6.2, accepts the physical layout of the target RICA core as well as the abstract netlist generated by the scheduler. It performs final register allocation as well as per step functional cell allocation, with the physical layout in mind. The main objectives are congestion removal and wire-length minimisation. The final output is a netlist format amended with path information per connection.

Bitstream Generation Based on a routed netlist, the bit-stream generator produces the final bit-stream to be loaded on the core. This tool includes the software side of the path-encoding, which is the configuration compression system. It forms the encoded configuration fragments and populates with those the memory rows accordingly. Although part of this work, the bitstream generator is not explicitly covered here, since its function is described in chapter 5, which explains the path-encoding and configuration context partitioning.

Simulator A cycle accurate, Transaction Level Model of the architecture, which allows the programmer to simulate the application. Depending on the level of detail of the input netlist, the simulator can check for path delays and report timing violations. It includes a detailed representation of the memory subsystem and thus can track the dynamic behaviour of memory access. The simulator can be used to get detailed timing analysis, switching activity and power estimation, utilisation trace graph, per step core dumps and memory dumps. Details of its implementation are found in section 7.4.

Emulator A behavioural model of the architecture, which provides fast execution for rapid prototyping. It provides similar debug facilities to the simulator, such as memory and core dumps and timing estimation. In addition to that, the emulator offers a virtual video frame buffer which can be used for testing multimedia application running on RICA as well as visual cues for debugging memory access patterns. The emulator is a separate project not covered by this work.

7.1.2 Backend

The following tools are part of RICA's physical design flow, suitable for architectural design exploration as well as final production of the core setup.

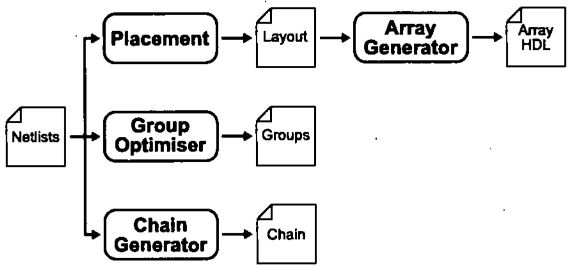


Figure 7.2: Toolflow overview: Backend

Physical Placement Produces the physical layout of the core. RICA is a heterogeneous architecture. The physical placement of the cells can impact the 'routability' of the core. The physical placement tool accepts a set of netlist from the target application domain. It analysis the contexts in those netlist and derives a suitable layout that gives satisfactory 'routability' across the range of tested applications. The main objectives are similar to the routing-allocation tool, but here, the tools is free to place each cell to any location on the grid of the array and does so based on an average estimation across several steps, rather than one step at the time. Because of this input netlist dependent behaviour, the physical placement will favour netlists of the input application domain, although tests, presented in section 7.6.1, show that the generated layouts are fairly generic.

Group Optimiser Defines the configuration context partitioning, which determines which group each cell belongs to. Similar to the physical placement, the partitioning derives from an analysis of a set of netlists from the target application domain. More details are found in section 7.6.3.

Chain Generator Produces the dictionary of the chain encoding. As with most of the RICA backend tools, the chain generation is based on a given target application domain. This tool is covered in section 7.6.4.

Array Generation An automated array generation tool, based on an input physical layout. It's a translation tool from the RICA netlist architecture definition to a top level Verilog file with the instantiation of the individual modules of the design.

The rest of this chapter describes those tools that have been developed as part of the work presented here. More specifically these include the routing-allocation, simulator, physical placement, group optimiser, chain generator, while the array generator and bit-stream generator are omitted.

7.2 Toolflow walk-through example

This section gives a brief overview of the compilation and configuration generation process by example, which form the basic steps of the RICA frontend.

Entry C code The following C code snippet shows a simple kernel, operating on an array named `input` and storing the results to an array named `output`. These arrays sit on the global scope and thus will be stored in memory. The code includes an expression that is intentionally chosen here to show how the compiler eliminates redundant operations. This is referring to the `i>100` within a for-loop where the counter `i` is in the range of `[0,99]`. This expression is always false and thus the condition always resolves to the right-handside expression of `(a - input[i])`.

```
unsigned i=0;
for(i=0;i<100;i++)
{
  int a = input[i] * input[i];
  int b = i>100 ? (input[i-100] + a) : (a - input[i]);
  int c = i%2 ? a : b;
  output[i] = c * c;
}
```

Compilation After compilation, the resulting RICA assembly for the above C code snippet is as follows:

```
loop_label:
  CONST    out=r4                          conf=!_input
  RMEM     out=r3 in_addr=r5 in_off=r4 conf='RMEM_SI
  MUL      out=r4     in1=r3    in2=r3 conf='MUL_SIG_SI
  ADD      out=r3     in1=r4    in2=r3 conf='ADD_SUB_SI
  LOGIC    out=r0     in1=r6    in2=r7 conf='LOGIC_AND_SI
  MUX      out=r0     in1=r4    in2=r3 conf='MUX_SEL_IN1_IF_NEZ sel=r0
  MUL      out=r0     in1=r0    in2=r0 conf='MUL_SIG_SI
  CONST    out=r3                          conf= !_output
  WMEM      in=r0 in_addr=r5 in_off=r3 conf='WMEM_SI
  CONST    out=r4                          conf=1
  ADD      out=r6     in1=r6    in2=r4 conf='ADD_ADD_SI
  CONST    out=r3                          conf=4
  ADD      out=r5     in1=r5    in2=r3 conf='ADD_ADD_SI
  CONST    out=r4                          conf=100
  COMP     out=r0     in1=r6    in2=r4 conf='COMP_EQ_SI
  CONST    out=w33                         conf=loop_label
  JUMP addr_in=w33 nl_out=OPEN cond=r0 conf='JUMP_IF_EQZ_H
```

where the redundant operation described above is optimised out.

In the compiler terminology, the above set of instructions represents a 'basic block'. A basic block is a sequence of instructions between branch points, designated by labels. The 'entry' label of this basic blocked is `loop_label`, while the 'exit' is only implicit here. The assembly has a few more basic blocks that are not shown here, which perform thinks like register initialisations and other house keeping.

The RICA assembly format is fairly self explanatory, starting with an instruction name and followed by the output and input arguments of the instruction. Registers are represented with 'rN' and immediates with 'wN'. In the assembly representation, data is passed on from one instruction to the next via registers. However, as seen later on, most of these registers are converted into wires when mapped onto RICA. Unlike conventional ISAs, each instruction specifies modes of operation, which are passed on via the `conf` input argument.

Notice that the jump-to address (`addr_in` argument) used by the `JUMP` brunch instruction, is the `loop_label` label, indicating an execution loop. The branch condition for the `JUMP` instruction is stored in register `r0`, which is the result of a single-integer equal comparison operation, indicated by the macro `'COMP_EQ_SI`. For more information on the various operation modes of each instruction cell, refer to Appendix C. This comparison operation corresponds to the `i<100` comparison of the for-loop in the C source code, though it has been converted into a do-while expression, hence the equal operation, rather than smaller-than. The loop counter here is register `r6`.

Scheduling The assembly is passed on to the scheduler that produces the abstract netlist, which is composed of a sequence of steps. The RICA netlist format is described in section 7.3. The above basic block forms a single configuration context (step), which is represented in the RICA netlist format as follows:

```
step[%loop_label]
{
  .reg[2].in = add[1].out;
  .const_32b[1].conf = 4;
  .add[1] { .in1=reg[2].out(length=1.0); .in2=const_32b[1].out;
          .conf='ADD_ADD_SI; }
  .const_32b[5].conf = 4;
  .reg[5].in = add[4].out;
  .rmem[0] { .conf='RMEM_SI; .in_addr=reg[5].out;
          .in_off=const_32b[5].out; }
  .mul[1] { .conf='MUL_SIG_SI; .in1=rmem[0].out; .in2=rmem[0].out; }
  .add[3] { .conf='ADD_SUB_SI; .in1=mul[1].out; .in2=rmem[0].out; }
  .reg[6].in = add[2].out;
  .reg[7];
  .logic[0] { .conf='LOGIC_AND_SI; .in1=reg[6].out; .in2=reg[7].out; }
  .mux[0] { .conf='MUX_SEL_IN1_IF_NEZ; .in1=mul[1].out; .in2=add[3].out;
           .sel=logic[0].out; }
  .mul[0] { .conf='MUL_SIG_SI; .in1=mux[0].out; .in2=mux[0].out; }
  .const_32b[2].conf = 404;
  .wmem[0] { .conf='WMEM_SI; .in=mul[0].out; .in_addr=reg[2].out;
          .in_off=const_32b[2].out; }
  .const_32b[3].conf = 1;
  .add[2] { .conf='ADD_ADD_SI; .in1=reg[6].out; .in2=const_32b[3].out; }
  .const_32b[4].conf = 4;
  .add[4]  { .conf='ADD_ADD_SI; .in1=reg[5].out; .in2=const_32b[4].out; }
  .const_32b[6].conf = @loop_label;
  .const_32b[0].conf = 1;
  .reg[31].in = add[0].out;
  .add[0] { .conf='ADD_SUB_SI; .in1=reg[31].out; .in2=const_32b[0].out; }
  .jump[0] { .conf='JUMP_IF_EQZ_L; .addr_in=const_32b[6].out;
           .cond=reg[31].out; }
}
```

In case there are not enough resources to map the basic block into one context, the scheduler would split those up into a sequence of smaller steps. Unlike the assembly, the netlist explicitly specifies the instances of each instruction cell in use, although these will change later on during allocation. The majority of register transfers are converted here into wires connecting the various instruction cells together, into operation chains, as seen on figure 7.3. In fact, all register instances present in the assembly are ignored and the scheduler infers registers as required, which is why the instances do not match the assembly. For instance, the for-loop counter was `r0` in the assembly, but is `reg[5]` in the resulting netlist and will yet again be relocated into something else during allocations. The tools provide debugging facilities to track down variables throughout the toolflow.

In the example generated netlist, the scheduler has performed an in-place optimisation / simplification of the branch condition and loop counter datapath, also show on the upper-left corner of figure 7.3. The initial counter increment and comparison to 100 has now been replaced with a shorter equivalent version of decrement and check to 0 performed by the jump cell, which does not need a separate comparator instruction cell. Furthermore, the jump condition is fed directly by the register that holds the counter, thus forming the shortest possible path.

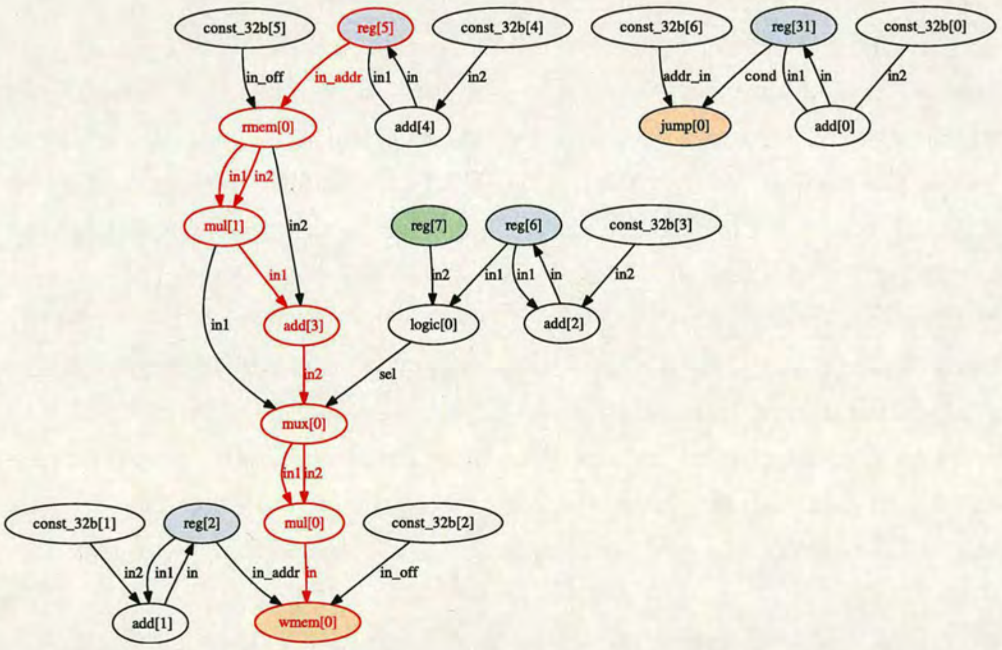


Figure 7.3: Example kernel: Data Flow Graph

The critical path is highlighted with red. There are only 5 instantiated registers in this step, where register `reg[7]` is an input, while registers `reg[2]`, `reg[5]`, `reg[6]` and `reg[31]` are both inputs and outputs to this step.

Allocation and Routing The generated netlist is passed on to the allocation and routing tool that maps each step into the physical resources of the array. The allocator chooses the appropriate physical instance for each instruction cell, while the router forms the physical paths connecting the cells together, as see on figure 7.4, which is the live view from the allocation and routing tool. In this view, the arrows represent the physical routing channels of the array, the black boxes represent the switch-boxes while the smaller colourful boxes the instruction cells. Straight lines are used to show the logical connections between the cells. The colours of the physical routing channels as well as logical connection lines match those of the driver cells. Cell instances and routing channels that are not used are not shown.

The instance indexes shown here are those of the input netlist, which helps maintain the association between the input netlist and the resulting mapping. However, these may not be the same as the physical instance indexes of the chosen cells, which can be viewed by selecting the alternative view in the viewer. The small sub-data-path responsible for the counter update and branch condition, is mapped here to the lower middle part of the array, where the `jump` instruction cell is situated. There one can see the counter `reg[31]` with a feedback to an adder `add[0]`, as well as driving the condition of the `jump[[0]` branch unit. `const_32b[0]` holds the constant 1 used for decrement, while `const_32[6]` the jump-to address, as can be seen by the above netlist. The reading and writing to the memory, done by `rmem[0]` and `wmem[0]` respectively, can be seen at the upper part of the core, where the memory interface cells are located. The rest of the graph is mapped in a way to minimize wire-lengths.

Bitstream generation The routed netlist is then passed on to the bitstream generation tool which produces the final configuration contexts to be loaded into the core's program memory. This tool path-encodes the connection tracks, assembles the path packets, appends the addresses and populates the appropriate 'groups' in every 'row' of each step. The resulting configuration

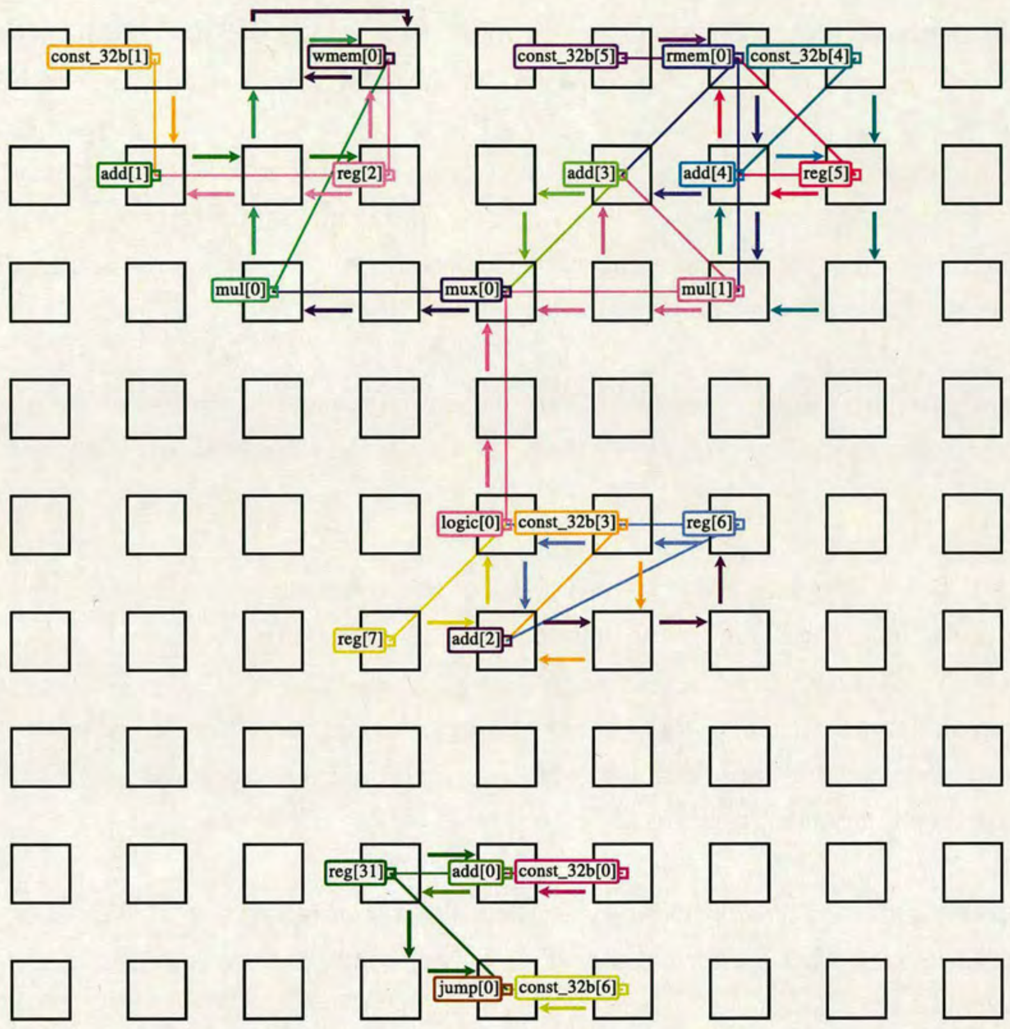


Figure 7.4: Example kernel: Routed

contexts look like figure 5.17 in section 5.13. The tool offers several output choices, including hex, verilog test vectors and a C array format for modeling.

7.3 RICA Netlist format

The RICA platform is a template rather than a fixed design, with the presented RICA prototype being an embodiment. This is made possible by the use of an abstract description of the various aspects of the design as well as the sequence of configuration contexts to be executed on the array. This description, which is defined as part of this work, is referred to as the 'RICA netlist' and serves as the common language/representation used by the entire RICA software toolflow, as seen on figures 7.1 and 7.2. This common abstract representation, allows the tools to become architecture agnostic in terms of resource mix and layout. The RICA netlist is composed of the following 'fields':

Architecture field Describes the various components of the architecture. It contains the *cell* and *layout* subfields as well as the alternative abstract *row* field.

Cell field Lists the available cell types of the architecture and for each cell type includes the definition of its port types and configuration size.

Layout field Provides the instantiation of the cells and thus specifies the instance count for each cell type as well as the switch box terminal port each of the cell instance ports are associated to.

Row field An alternative abstract field for instantiating cells, which does not provided layout information. This only specifies the instance count for each cell type.

Data field Includes the global data memory containts of the program

Sequence field Is a list of *step* fields that describe the entire configuration sequence.

Step field A connection oriented description of a configuration context. It can optionally include routing information, describing the detailed paths the connections use.

The detailed RICA netlist format can be found in Appendix B.

7.4 RICA TLM simulator

7.4.1 Transaction Level Modelling and SystemC

Transaction Level Modelling(TLM) offers a high level of abstraction view of the target design, with emphasis in the interaction between the various components of the system. As such, it allows easy and fast prototyping as well as early modeling of the underlying system. In case of a hardware design, early modeling of the system can then engage the development of the associated software stack much earlier in the design process, thus allowing parallel development of the two. This offers a huge improvement over conventional serialised methodologies, since the hardware spec can be forged to better serve the needs of the software as well[59].

TLMs are usually written in high level languages, such as SystemC. SystemC is a general modeling language based on C++[60], which allows a broad scope of modeling levels of abstractions, including TLM. The reference implementation comes as a C++ library that allows a developer to create an executable model of the target system.

Although based on C++, SystemC adds hardware oriented concepts, such as concurrency. A SystemC model starts with the *elaboration* phase, where the individual components of the design are instantiated and the communication channels between them established. In this regard it's similar to HDLs. After the elaboration phase, the simulation is engaged, at which point no futher instantiations are possible. The system's skeleton it thus fixed. However, since one can describe almost any kind of construct, it's possible to describe flexible architectures, such as reconfigurable designs, albeit with some modifications. Section 7.4.3 describes one such modification.

7.4.2 The simulator design

The TLM simulator is more or less a software equivalent of the RICA hardware implementation. Some aspects are simplified, such as the interconnects, to improve flexibility and simulation performance, while others are close to a one-to-one mapping, such as the data memory arbiters.

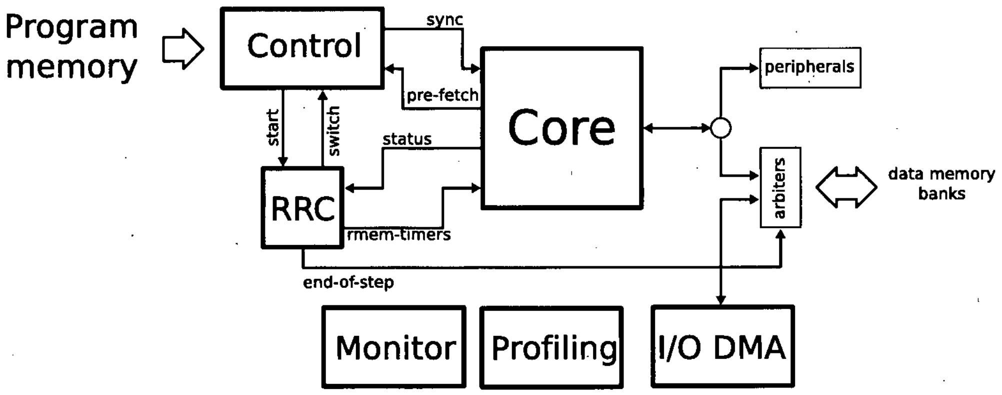


Figure 7.5: Simulator overview

The *Core* encapsulates the RICA cells and the interconnects. During the elaboration phase, the cells are instantiated and bound to the interconnects. The cell type and instance count is read from the architecture definition of the netlist file (see Appendix B). The RRC and Control units orchestrate the context execution, while the Monitor and Profiling units provide introspection.

The *interconnects* are an abstract representation of the real routing resources. Having an abstract representation allows the simulation of various topologies or types of routing resources, from crossbars to island based and 2 Dimensional arrays. From a TLM point of view, these topologies only differ on the number of intermediate hops a given connection needs to go through. This is controlled here via a configurable set of delays. More complicated concepts, such as routing conflicts and arbitration, can be implemented on top of this design, that would allow the simulation of things such as shared-bus topologies or message passing protocols. A more detailed description of the interconnects implementation in SystemC can be found in section 7.4.3.

The RRC is almost a direct implementation of the corresponding hardware module as seen in section 6.2. The *Control* unit implements part of the program memory interface of the hardware implementation. The Control unit emulates, among other things, the pre-fetching mechanism. An overview of its implementation is shown in figure 7.6.

The condition to pre-fetch the next configuration context is given by the

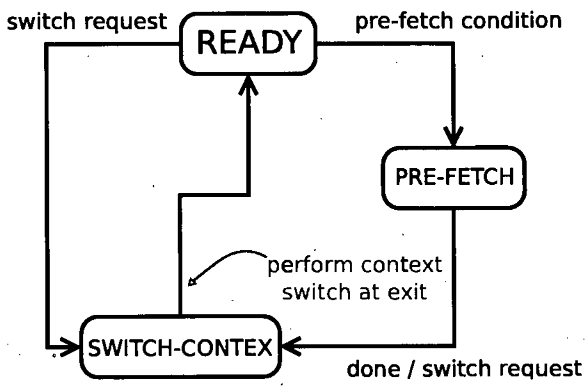


Figure 7.6: The pre-fetching mechanism

Core itself and more specifically the flow control unit (jump cell), as seen in figure 7.5. If any of the pre-fetching conditions described in section 6.3 are met, the Control unit will move to the PRE-FETCH state of its FSM and will remain there until either the pre-fetching is complete or a new context switch is requested. The core will switch to the next context when both pre-fetching is complete and a new context switch requested. The order is not important.

7.4.3 Configurable interconnects in SystemC

Reconfigurable interconnects are able to dynamically alter the physical route of a signal and thus establish a number of alternative connections. The possible combinations depend on the type and topology of the interconnects as seen in section 3.3. In HDL, such interconnects will involve a number of multiplexers. When it comes to high level modelling of such flexible structures, it is desirable to capture as many of those types and topologies as possible, thus maximising the design space exploration. A multiplexor is a low level implementation detail and thus not suitable for a high level model.

In SystemC, binding of all ports to interfaces is done during the *elaboration* phase and thus cannot be changed during simulation. In fact, ports left unbound cause run-time errors and the simulation aborts. This is not very different to HDL. All connections of a system are explicitly defined and form the system level wiring of the design. An abstract implementation of a reconfigurable interconnection needs to work around this practical limitation.

From a bird's eye view, the reconfigurable interconnects provide access points to the units that are attached to them and an interface to allow altering the connections. The method chosen here uses an input-centric approach, where a terminal input port 'decides' which terminal output port of another unit to establish a connection with. Compared to an output-centric approach, this is a more natural and simple representation, since it describes a one-to-one association rather than a one-to-many, in the case of a fan-out.

The interconnects object encapsulates a 'sea' of *channels* and provides means to bind terminal ports to those channels and retrieve a channel by ID number. The channels are created during the *elaboration* phase, as ports are registered to the interconnects. Each terminal port gets its own channel which it either listens to or writes to. The output channels, those registered to an output terminal port, remain fixed throughout the simulation. The input channels are not used to establish connection, but only provide a way to assign *immediate* constant values to ports, if the architecture supports such a thing. By default, an input terminal port listens to its channel. A connection reconfiguration will change that association to an output channel of an output terminal port.

The connection establishment as well as the connection itself, can be coupled with a number of properties to emulate possible arbitration logic, propagation delays, congestions, et cetera. Through this abstract representation one can emulate a wide variety of topologies and communication protocols, providing an excellent design exploration tool.

7.4.4 Features

Other than a high level model of the RICA architecture, the simulator offers a number of useful features that can aid the software developer in profiling and optimising the target application. These are:

Execution timing report Since the simulator is circle accurate, it is able to report the exact execution time of the entire program, as well as the configuration latency. It provides a detailed report of the time each step took, as well as how many times each step was executed and how many of those times it was fetched or pre-fetched from the RAM. Along with the required number of configuration bits per step, this allows one to calculate the overall program memory bandwidth requirements

Debug outputs To aid the debugging of an application running on RICA, the simulator provides a number of debugging outputs, such as core dump of all register and cell output contents, as well as the data memory contents. These debug outputs follow the execution flow and report the values on a per step bases.

Dynamic model of RAM subsystem The simulator has a detailed view of the arbitration of the data memory subsystem. It can thus measure the stall times and performance degradation of accessing the same bank multiple times. It can thus offer the developer means to improve memory alignments and avoid such congestions.

Switching activity The simulator can track the switching activity of every cell and every part of the system, such as the fetch unit and data memory arbiter. Using power and timing values from the actual hardware, the simulator can calculate the power consumed by the device whilst running the target application. A developer could then optimise the design for power as well as speed. The core switching activity can be viewed in the core utilisation output of the simulation, as in figure 7.7.

Detailed time charts Having a detailed view of the core, allows the simulator to provide a graph of the activity of each cell in time. These 'time-charts' show how long and at what point it time each cell is active, thus providing a simplified wave form for each step. An example time-chart can be seen on figure 6.8 in section 6.3.

Utilisation trace graph The simulator can produce dynamic trace of the core's utilisation as well as the data memory utilisation. Figures 7.7 and 7.8 show the output of the simulator for the core and memory utilisation trace of the MP3 libamd application running on RICA. The green bars represent in figure 7.7, represent the static core utilisation, while the red the switching activity.

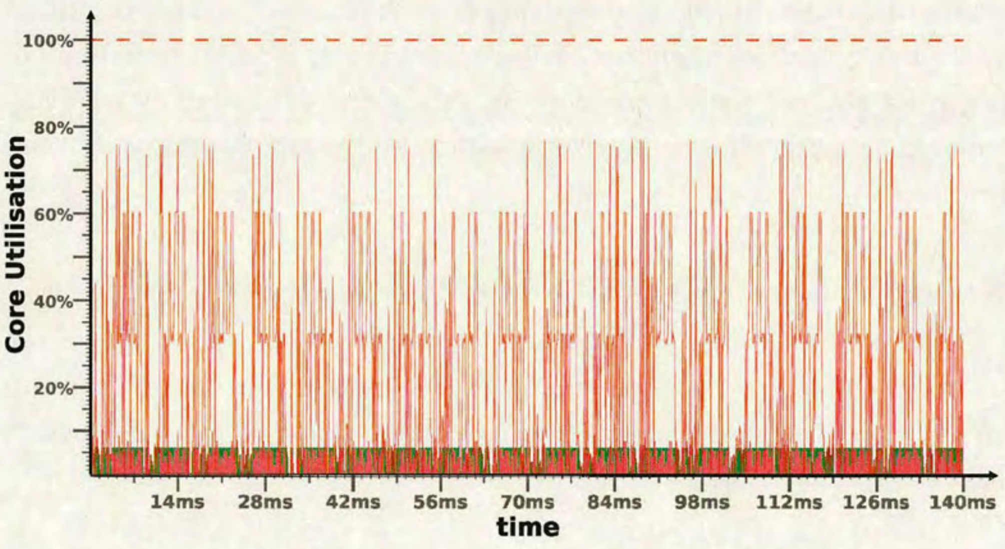


Figure 7.7: Core utilisation trace for libmad

From these figures, one can see how the core iterates around the kernel steps, which represent high utilisation and high memory activity, with setup steps of low activity in between.

7.5 The Multi-Objective Multi-Threading GA Framework

The various optimisation and mapping tools described in section 7.6, are based on a genetic algorithm framework that is part of the research of this thesis, referred to us MoMtGA. The use of this multi-objective GA framework makes it easy to deploy a variety of problem solvers. Implemented in C++, it is based on

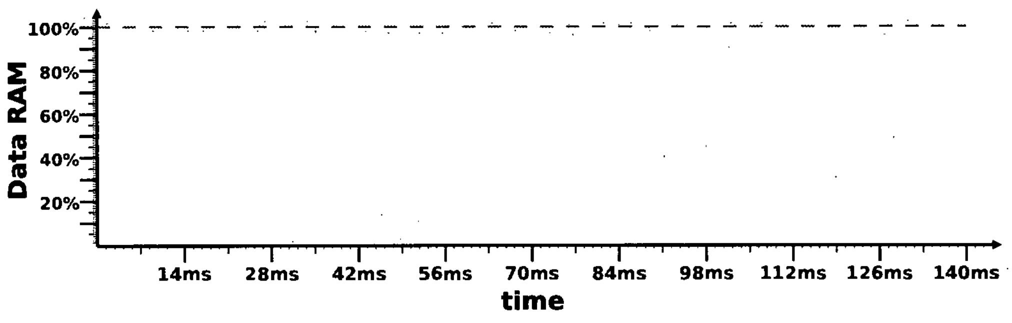


Figure 7.8: Memory utilisation trace for libmad

an object-oriented programming model. The entire framework is autonomous and provides 'hooks' for the cost and chromosome decoding functions.

MoMtGA is lightweight and exploits the intrinsic parallelism present in GAs, where individuals can be evaluated independently and concurrently to each other. The framework uses a batch processing engine suitable for fine grain tasks, implemented using POSIX threads [61], thus being able to harness the parallel processing power of modern multi-core systems. Other known GA libraries on the public domain, have a serial implementation [62, 63], while some come with MT [64] or MP support [65, 66, 67] as well. Unlike those ones with parallel support, MoMtGA's MT support is transparent to the implemented algorithm. Of course, as in all cases, the end performance of the algorithm depends on the implementation of the cost function, which is application dependend.

There are no special purpose genetic operators used or required. The chromosomes are bit sequences and partitioned into parameterisable clusters that can be used to map to individual traits of the application's phenotype. The breeding process involves the creation of 4 offsprings, 2 of which are produced using single-point crossover and 2 using uniform recombination, as shown in figure 7.9. The best 2 offsprings are inserted into the gene pool after a tournament.

The natural selection is based on fitness proportionate selection using *roulette wheel* and *ranking* [35]. The individuals are ordered based on their fitness value. In addition to do this, each individual receives a 'ranking score' which is roughly proportional to its position in the population list. The described implementation assigns a ranking score equal to the population count starting from the highest ranking individual and one less for each next individual with smaller fitness score. Thus, individuals with the same fitness score will have the same ranking score. This 'proportionate' ordering provides fair selection probability later on when the roulette wheel is used.

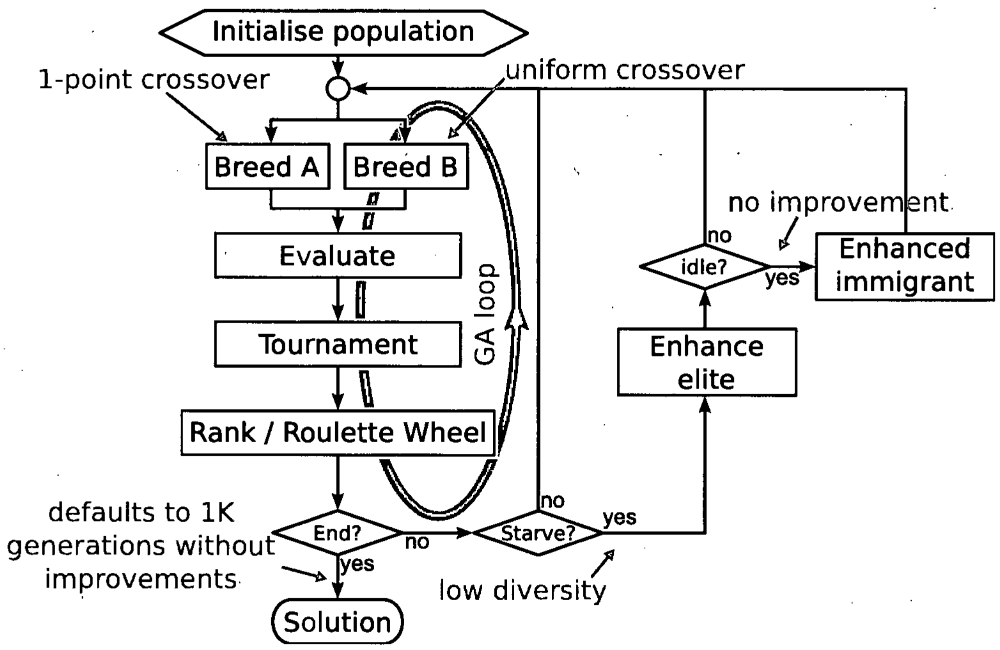


Figure 7.9: GA Framework flow chart

The roulette wheel selection, as the name implies, resembles a roulette wheel where each individual reserves a portion of the roulette's slices according to its ranking score. The higher the ranking score the greater the proportion assigned to the individual. As a consequence, the roulette's wheel slice count is equal to the sum of all ranking scores. The natural-selection is done by randomly selecting a slice from the roulette's wheel range, i.e. $slice_{count}$ to 1, and then peeking the individual that the slice belongs to.

The GA framework supports priority-based multiple objectives. This is done using a fitness vector, or composite, where each element is the fitness of one objective. Given two vector fitness metrics F_a and F_b, each with N objectives, where $F_a[0]$ represents the highest priority objective and $F_a[N-1]$ the lowest, for F_a to be greater than F_b, there must exist at least one objective n, where $F_a[n] > F_b[n]$ and for every other objective i in the range 0 to $n-1$, $F_a[i] = F_b[i]$. Same applies for the $F_a < F_b$ condition. A shared fitness can be represented by simply combining several objectives in one fitness component, with appropriate weighting. The vector fitness provides a wide variety of combinations of priority and shared based individual fitness metrics. The common comparison operator conditions of a vector fitness are shown below.

$F_a > F_b$ when $F_a[n] > F_b[n]$, $n \in$ [0,N-1] and $F_a[i] = F_b[i]$, $\forall i \in$ [0,n-1]
$F_a < F_b$ when $F_a[n] < F_b[n]$, $n \in$ [0,N-1] and $F_a[i] = F_b[i]$, $\forall i \in$ [0,n-1]
$F_a = F_b$ when $F_a[n] = F_b[n]$, $\forall n \in$ [0,N-1]

The usual parameters, such as mutation and crossover probability as well as population size, are parameterisable. The default mutation and crossover values are 10% and 70% respectively, chosen empirically after numerous experiments. The population size varies depending on the size of the search space, which is proportional to the chromosome length. The GA framework supports the notion of elitism and provides a mechanism to overcome potential local optimum, as described below

7.5.1 Enhancement

As seen in figure 7.9, the GA framework described here is a hybrid one. It employs an enhancement function to overcome local minima, in case of a minimisation function. The algorithm branches out from it's normal iteration into the 'enhancement' section when the population's diversity is lower than a given threshold. The diversity can be measured by counting the number of identical elites in the population. The enhancement is applied to one of those identical elites, thus producing a variant of the original elite as described below.

The enhancement is a heuristic that works on traits and their corresponding genes. It thus evaluates direct modifications to the phenotype. The enhancement algorithm picks one random trait at a time and covers its entire range of possible variations, trying to find one that results in higher total fitness. It does so for every trait available in a random fashion. In a way, the individual undergoes a regulated mass mutation. The resulting individual may well be quite different from the original one, but usually resembles the original elite. The enhancement brings new gene combinations into the gene pool, which helps the GA to overcome local minima. Due to the randomness built into the enhancement function, applying it to the same individual may give different results each time. Therefore, one could introduce more than one of those enhanced elites to the population with different genotypes, all derived from the same original elite individual.

In most cases the GA evolves up to a given stage by its own until it can not find any better solution, at which point a cycle of subsequent enhancements, with evolutionary steps in between, commences. The algorithm retires when no further improvements are seen after a given number of generations. The enhancement function kicks in when the diversity drops bellow a certain threshold. The diversity is calculated by observing the ranking scores of the individuals. Individuals with the same fitness receive the same ranking. Thus a larger ranking scale indicates higher diversity and vice versa.

A further way to overcome local minima is to introduce random immigrants. As seen in figure 7.9, these are added when even the enhancements can't result in any improvements. The random immigrants rely on pure randomness and as such normally yield in low chances for improvements. A random immigrant will usually be rejected very quickly by the GA due to its low fitness score. In an attempt to prevent this from happening, the random immigrants are first enhanced, with the above enhancement function, before entering the population.

7.6 The RICA EA tools

Most of the backend tools of the RICA toolflow, that involve finding a solution, are based on the GA framework described in 7.5. Four of those tools are described here, which include the physical placement tool, the allocation and routing, group optimiser and chain generator.

7.6.1 Physical Placement for HDRA

In this context, physical placement refers to the positions on the sbox-array the functional cells are assigned to, as opposed to allocation, which refers to the association of an operation to a physical instance. In a heterogeneous core, the functional units can have different sizes. To maintain a balanced interconnection network with uniform delays across the array, the switch boxes are first laid out in an uniformly spaced grid, allowing enough space for the function cells to be placed. The logic cells end up occupying the empty space around the sboxes, as seen in figure 7.10. The highlighted part in this figure is a logic cell which ends up spanning across 3 rows of sboxes. Other smaller cells will end up occupying less that the height of an sbox. This kind of arrangement may create a variation between the various cell instances. However, even if those variations end up being large enough to cause problems, they can be included as part of the delay estimations of the compiler/scheduler and mapping tools.

Most of the aspects of the placement algorithm described here are generic and apply to a wide range of layout exploration applications. The few implementation dependent features that exist, exploit some intrinsic characteristic of the target platform which is described below. The placement tool accepts an abstract representation of the configuration *contexts* of an application and a description of the resource count and type of the target platform. This representation is referred to here as *temporal netlist* or simply *netlist* as seen in figure 7.11. Optionally, the netlist may include the position of some of the logic units within the array. This allows us to shape the layout based on some pre-defined characteristics. Almost every aspect of the design is parameterisable.

The target reconfigurable platform is the RICA prototype design, which consists of an array of heterogeneous logic units in a 2D toroidal mesh topology of interconnected switch-boxes (sbox) as seen in figure 7.12. The array consists of a wide variety of logic units, such as boolean and simple algebraic arithmetic operators, storage units, decision and predication units, memory interface units as well as control flow units. The mixture of those different types depends on the application domain the platform is targeted for.

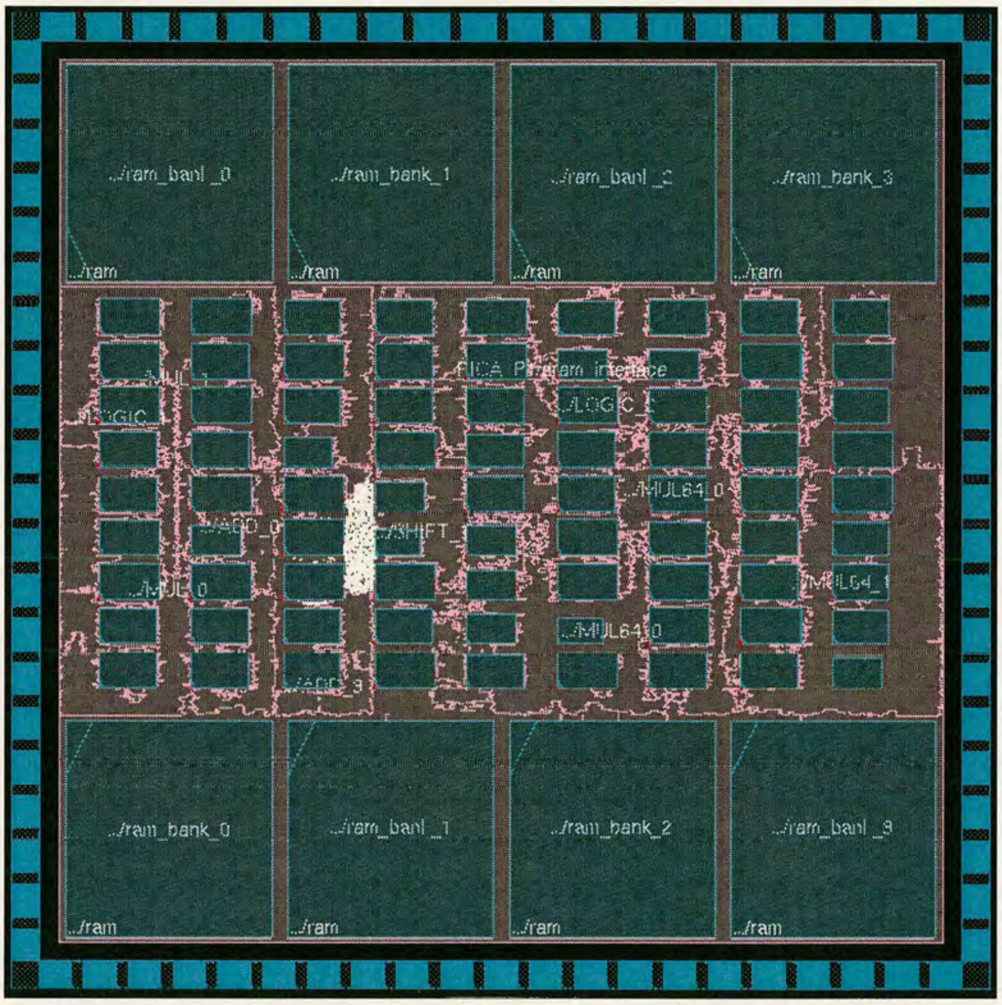


Figure 7.10: RICA prototype physical placement

As seen in figure 7.11, the architecture definition field of the netlist, specifies the available resource count and type. For a given operation described in one context, the placement tool is able to use any of the available resources that can perform that operation. In other words, the placement algorithm is allocation aware. Nevertheless, some logic unit types are not re-allocatable, such as storage units, in which case the placement algorithm has to use the same instance as described in the input configuration context

An sbox is assumed here to have 4 pairs of uni-directional network ports, 2

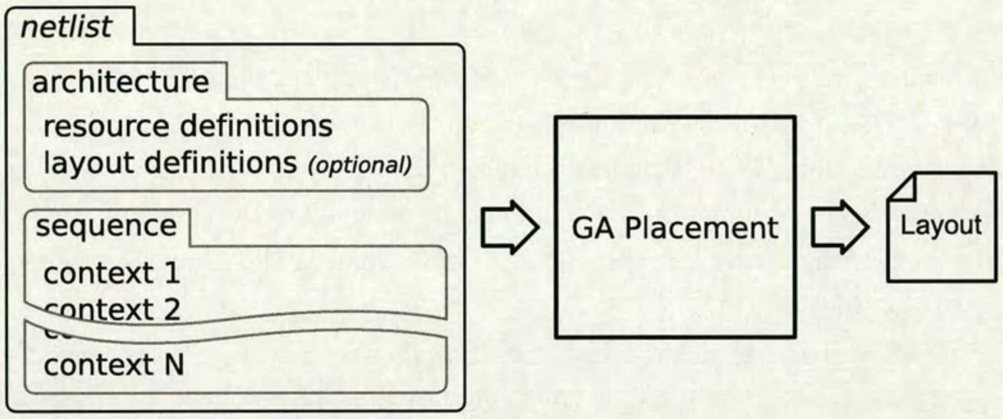


Figure 7.11: Placement tool overview

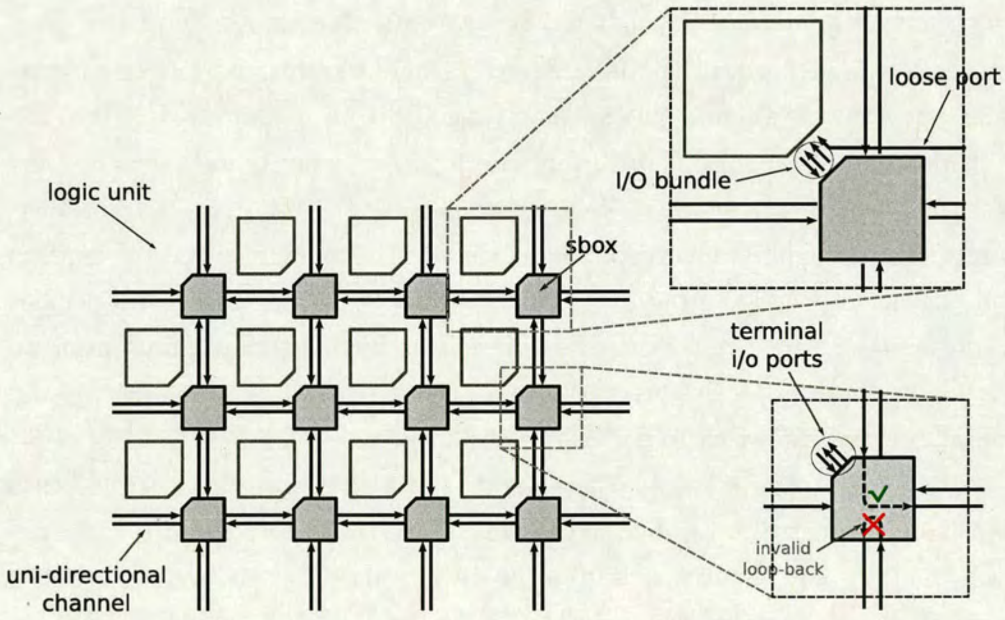


Figure 7.12: 2D toroidal mesh topology

output and 1 input terminal ports, which connect to the attached logic unit(s). Furthermore, a logic unit has up to 1 output port, however it may have more than 2 input ports, thus requiring more than one sbox. A different setup could be used without affecting the applicability of the algorithm. The sbox can connect every output port of one *i/o port pair* to one of the input ports of another pair only as seen in figure 7.12. Hence, loop-back connections are not allowed. This is one of the imposed restrictions of the target platform. Because of this restriction, if one logic unit connects to the input terminal port of an sbox, the output terminal ports should also be assigned to the same logic unit. Otherwise a connection between the logic units sharing the same sbox would not be possible.

The above features of the target platform can be used here to reduce the total search space, thus improve convergence time. This is done by dividing the logic unit ports into two categories. The *i/o bundled ports* and the *loose input ports.* For instance, if a logic unit has one output and 4 input ports, 2 of those inputs are *bundled* with the output and assigned to one sbox. The remaining 2 input ports are left as 'loose' and may occupy any of the 'free' output terminal ports in the lattice. An output terminal port is considered 'free' when not in use and the input terminal port of its sbox is also free.

Unless otherwise specified, any of the logic unit input ports can be part of an i/o bundle or be a loose one, in which case the final port allocation is controlled by the placement algorithm. The algorithm handles this by dividing the allocation process into 2 stages. In the first stage, the algorithm decides where best to place the i/o bundles and which sbox output terminal ports to use for the various loose ports. While in the second stage it shuffles around the physical ports trying to find the best allocation.

To simplify explanation, let's assume that every logic unit has 3 input ports and 1 output port. Let $L[i]$ be the i-th logic unit of our array and $L[i] : input[p]$ its p-th input port, while $L[i] : output$ be its output. Similarly, $sbox[y][x]$ is the sbox in the (y, x) position and $sbox[y][x] : term_{out}[t]$ and $sbox[y][x] : term_{in}$ are its t-th terminal output and terminal input ports respectively. If N is the number of sboxes in the array, then there are $N!$ permutations available for the logic output ports. As explained above, every output port bundles 2 of the available inputs into an *i/o* bundle ($L[i] : bundle$) and thus 1 input remains loose ($L[i] : loose[0]$).

The final port mapping is decided in the second stage. A possible outcome of the first stage could be the following:

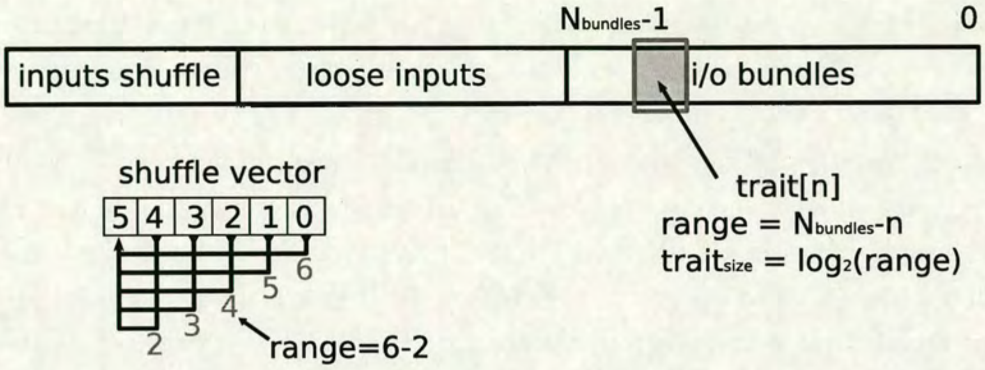


Figure 7.13: Chromosome encoding

$L[4]$: *bundle* assigned to $sbox[3][2]$
$L[4]$: *loose*[0] assigned to $sbox[1][5]$: $term_{out}[0]$

The *bundle* reserves all terminal ports of the sbox it is assigned to, preventing other logic units from using them. In the second stage, the final port mapping is decided by a *shuffle* vector controlled by the chromosome. A possible outcome could be:

shuffle vector: [0 2 1]
$L[4]$: *output* attached to $sbox[3][2]$: $term_{in}$
$L[4]$: *input*[1] attached to $sbox[3][2]$: $term_{out}[0]$
$L[4]$: *input*[2] attached to $sbox[3][2]$: $term_{out}[1]$
$L[4]$: *input*[0] attached to $sbox[1][5]$: $term_{out}[0]$

where $L[4]$: *input*[0] is here the *loose* port

As seen on figure 7.13, the chromosome is divided into 3 sections. The first section controls the positions of the *i/o bundle*, the second the *loose* input ports and the third is used during the second stage for the final input port mapping. Each section is a shuffle vector with variable length elements. Given an initial ordered layout vector, a shuffle vector repositions the elements of the layout vector by applying multiple swap operations on it. For an N sized layout vector there is an N-1 shuffle vector, as explained below. Each element of the shuffle vector defines a swap operation, where the operands are two layout elements. Let *swap*[*i*] be the *i*-th swap operation and *layout*[*i*] the corresponding element on the layout vector, which forms the first operand. The *swap*[*i*] controls the current *layout*[*i*] and can swap it with any layout element in the range of $N-1$ to i index, where choice i would leave it unchanged. Therefore, the 'range' of the *swap*[*i*] operation is $(N-i)$, thus requiring $\log_2(N-i)$ bits. Since this can not always be an integer number of bits, the result is always rounded up. The decoded value is 'folded' to the specified range. The Nth element of the layout vector has no swap operation associated to it, since there is no element left to swap it with. Hence why the shuffle vector has a size of N-1.

The effects of the shuffle vector have temporal dependencies. Thus the results of the *i/o bundle* are fed to the *loose* shuffle vector as an input. A single change in the genotype could result in several changes in the phenotype, which generally is something to be avoided, since it can cause instability during the evolutionary process. Unfortunately this interdependency is unavoidable, due to the inherent properties of the placement problem.

Cost functions

A DRA has a set of *contexts* that are meant to be mapped temporally and switch in various time intervals throughout program execution. A context is the physical description of connections between logic units along with their configuration status, if any. DRAs are programmable devices and as such can perform execution flow *branches* or *loops.* Hence, one or more contexts may repeat for several times within an execution loop controlled by a conditional branch. Logic units that tend to connect with each other more frequently than others, should get higher priority when making placement decisions. A post-processing analysis is made to collect statistical information for the connection frequency of all pair combinations. The ideal scenario would be to use run-time statistical data extracted from program execution. Because this is not always possible, due to the often dynamic nature of the processed data, a statistical analysis of the program sequence is made with some hinting for potential loops.

One of the objectives of the placement algorithm is to make the resulting layout easier for routing the mapped netlists. This objective is based on a congestion indication metric that uses overlapping routing regions as those seen in figure 7.14, here after referred as *routability* metric.

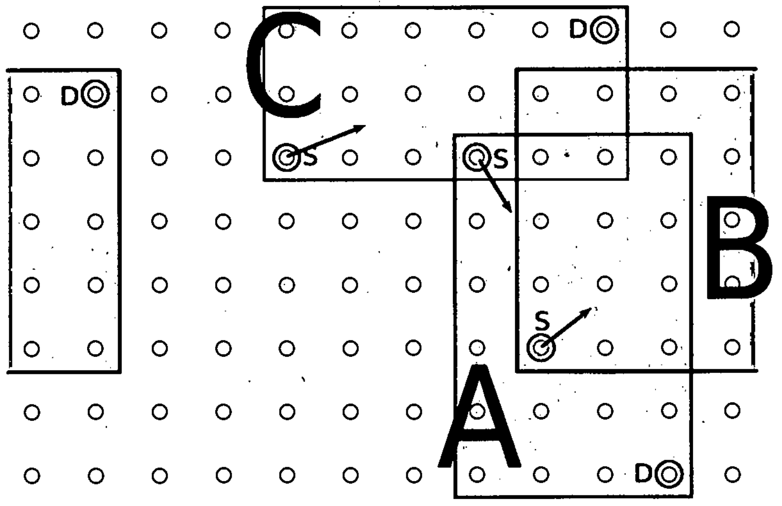


Figure 7.14: Overlapping Routing regions

A source (S) and destination (D) pair defines an optimum routing region, where paths from one node to the other have optimum lengths, given that they are *fully monotonic.* Fully monotonic paths are those that traverse in one direction in both axes, e.g. a path that goes East-South without ever turning West or North. During routing process, the routing algorithm will attempt to produce such fully monotonic paths, thus it is best to assume that most paths will be fully monotonic during placement evaluation. When two routing regions overlap, partly or fully, it is an indication that congestion may occur. The more regions overlap the higher the congestion probability. Reducing the number of these overlapping areas is thus a wise move.

Bearing in mind that each routing node (sbox) has two uni-directional channels per network port, two regions may overlap, but may not suffer from congestion due to their source-to-destination directions. Therefore there are 3 possible cases:

a both directional-axis are the same, then the regions conflict, b one of the directional-axis is the same, thus the regions can only conflict on the shared directional-axis and

c none of the directional-axis are the same, in which case the regions are considered conflict free.

When calculating the routability metric the above 3 cases are given different weights.

Another objective of the placement algorithm is to reduce the total *wire-length*. The actual wire length can not be known before routing is applied, but assuming that most paths will be fully monotonic, the minimisation of the optimum path lengths will suffice. Once again, due to the temporal nature of the configurations, the placement algorithm needs to prioritise the available connections based on their frequency of occurrence. The wire-length cost function applies a weight factor to each connection pair that is proportional to the frequency of that pair.

The third objective, *port-grouping*, refers to the distance of the input and output ports of each logic unit. To prevent them from being placed far apart, that would otherwise cause issues during floorplanning, the placement algorithm gives this objective the highest priority. For this to work, the algorithm needs a reference position to evaluate the distance of each port from that position. For those logic units that have an output port and thus form an i/o bundle, the reference position is the sbox where the bundle is assigned. For logic units that have only loose ports, the algorithm finds a middle point that's closer to all ports and uses that as the reference point.

Experimental Results

The target array for these experiments is an 12 x 12 dimensions with functional units composed of around 35% register, thus non re-allocatable, 27% boolean and algebraic operations, 20% decision and conditional/predication units, 17% program and data memory interface units, as well as 1 flow control unit for the execution of single threaded applications, as shown on figure 7.15. The input netlist used for the placement optimisation is from an MP3 audio decoder ported to the target platform. The netlist consist of around 2K contexts and contains a wide range of DSP operations.

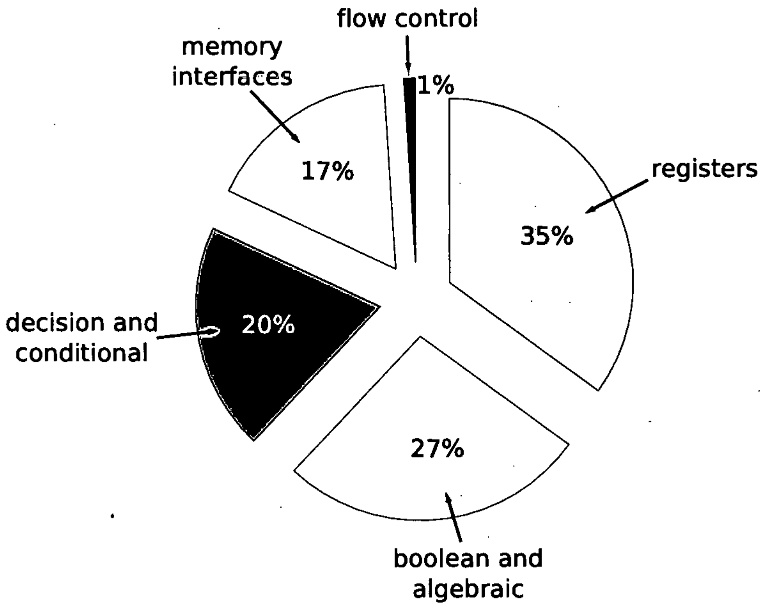


Figure 7.15: RICA array operations breakdown

GAs are known to be non-deterministic in nature due to the randomness they employ. Therefore each optimisation run may have different outcomes. As long as the final results have acceptable quality and small deviation, this variability is not an issue. The results on tables 7.1 and 7.2 show that the standard deviation (stdev) is fairly small. The following graphs on figures 7.16 and 7.17 show two diverse cases out of 10 runs. The graphs depict the normalised cost of each of the placement objectives of the best individual on every generation, throughout the placement optimisation process. The routability metric is measured in 'area units' while the wire-length and port-grouping in 'distance units'. For illustration purposes the the costs/metrics have been normalised and scaled to the same unit-base.

The first graph shows an optimisation run where the objectives improve progressively. This run also represents a case where the primary objective port-grouping receives better improvement than the secondary ones, routability and wire-length. The sudden change just before the 500th generation, is caused by the enhancement function that was applied at that stage, helping the GA to leap forward. Other cases where the enhancement function was applied can be seen with smaller improvements.

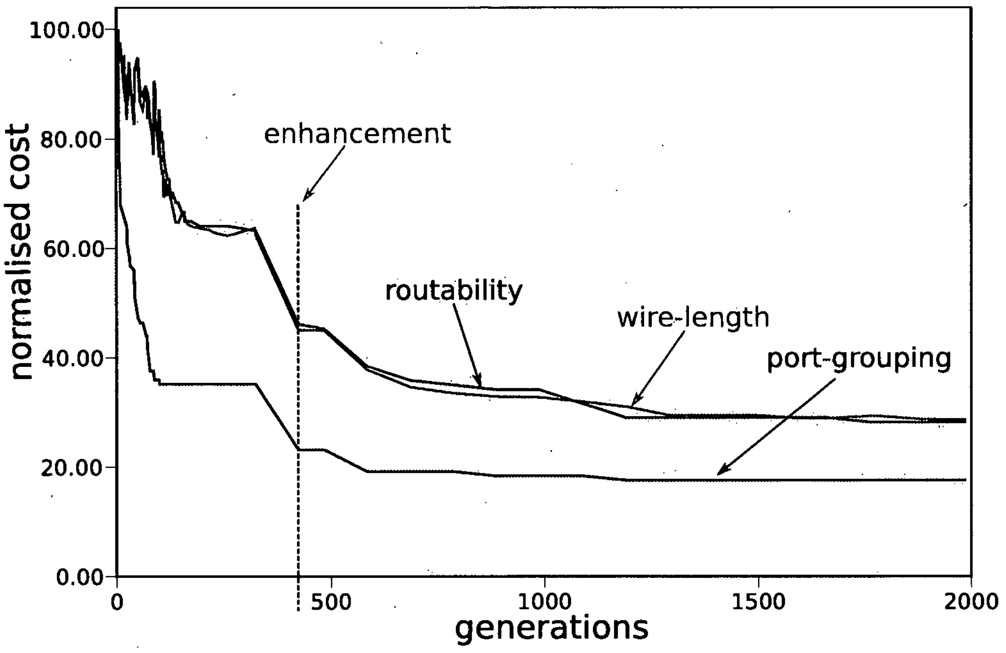


Figure 7.16: Normalised cost/generations

The second graph shows an interesting side effect of the prioritised multi-objective fitness. Since the port-grouping is the primary objective, its effect can 'dominate' over the secondary objectives in the calculation of the total fitness. An improvement on the primary objective will always result in a better total fitness score, even if the secondary objectives degrade. Figure 7.17 shows one such extreme case, during an enhancement process, where the secondary objectives degrade sharply.

The effect is temporary though, since successive generations smooth things out. This is because, even though the 'enhanced' elite has bad secondary objective scores, other individuals in the population have good secondary scores for those same objectives. After breeding, the good traits are inherited over

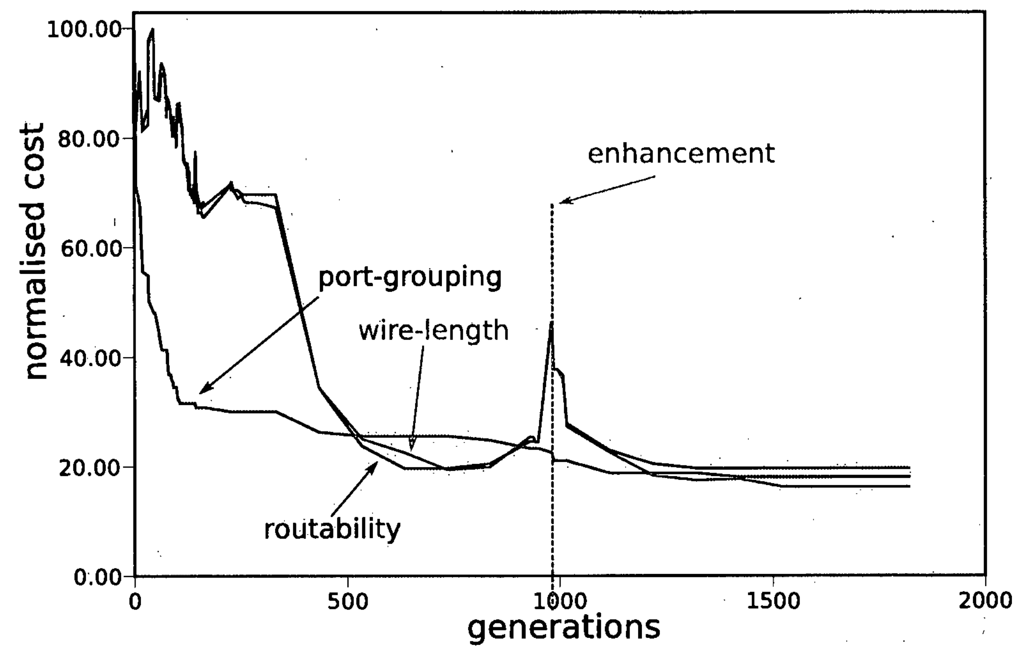


Figure 7.17: Normalised cost/generations

and the resulting fitness performs well on all objectives. A shared fitness does not exhibit such a behaviour, but at the same time it does not guarantee a high quality port-grouping, which is essential.

In order to test the quality of mapping, the resulting placements have been used to allocate and route a set of netlists. Figure 7.18 shows the quality metrics of the resulting final routed netlists, compared to a uniform layout of the heterogeneous resource, as well as the same netlists on a homogeneous array using ALUs. The uniform heterogeneous layout has the instances of each cell type spaced appart in roughly equal distances. The homogeneous array assumes that all functional units are the same ALUs which can perform any of the supported operations. Thus the allocation is free to choose any of them.

The 'optimum contexts' in figure 7.18, refers to percentage of the contexts of the entire program sequence that have all of their routing paths optimum, i.e fully monotonic. The 'wire-length quality' metric is normalised against the results obtained by the homogeneous array. The 'routability' metric refers to

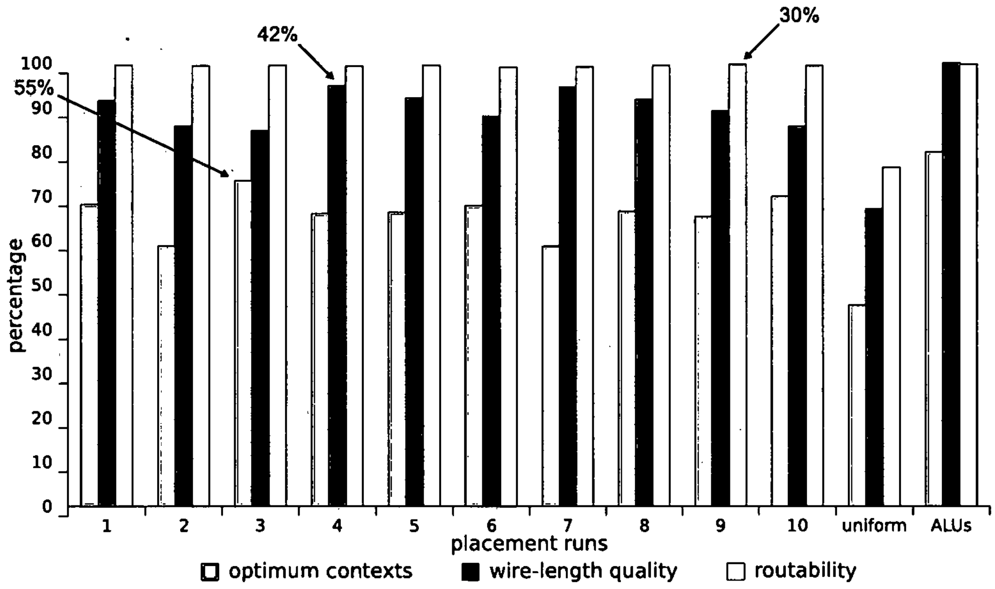


Figure 7.18: Placement quality

the percentage of successfully routed paths. In all cases the 'routability' is improved by about 30% compared to the uniform heterogeneous placement. The improvements in 'wire-length' vary from 26% to 42%, while the overall percentage of high quality contexts show an increase from 30% to 55%. Table 7.1 includes more analytically those results.

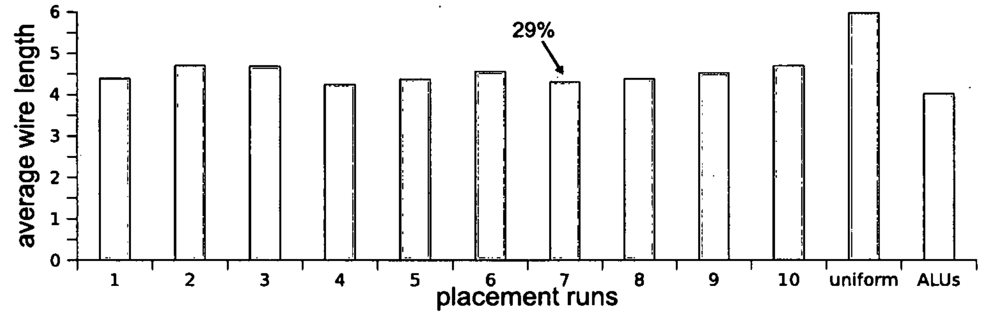


Figure 7.19: Average wire-length

Figure 7.19 shows the average wire length achieved for all runs compared to the uniform and homogeneous placement. The results show improvements ranging from 21% up to 29% compared to the uniform heterogeneous placement. Table 7.2 includes the results obtained by the allocating and routing the test netlist on the various generated layouts.

	optimum context	optimum paths	wire-length quality	routability
run1	68.12	96.06	91.57	99.52
run2	58.78	95.64	85.85	99.41
run3	73.49	97.21	84.75	99.52
run4	66.04	95.74	94.88	99.29
run5	66.36	95.71	92.21	99.52
run6	67.85	95.99	88.08	99.09
run7	58.64	94.54	94.63	99.16
run8	66.59	95.88	91.85	99.52
run9	65.41	95.72	89.18	99.70
run10	69.93	95.95	85.73	99.42
stdev	4.55	0.64	3.69	0.18
uniform	45.37	91.95	67.15	76.49
ALUs	79.95	96.18	100	99.71

Table 7.1: Placement quality results

Heterogeneous architectures attempt to reduce the overall area of the design by introducing specialisation. This is a design decision that trades flexibility, which affect the routability of the design, for silicon area. What the results show here is that it is possible to improve routability by carefully selecting the physical placement of the functional units in the array, without requiring to increase the routing resources. In many cases, the optimised heterogeneous layouts perform very close to what the the homogeneous layout achieves.

7.6.2 Routing and Allocation

The mapping process for each configuration context(step) is done in 2 separate phases, the *allocation* and the *routing*. The allocation is based on the routing-aware estimations, using the 'overlapping-regions, described in 7.6.1. As such, it has 2 objectives, the routability and the wire-length objective. The routing

	average optimum wire-length	average wire-length
run1	4.22	4.39
run2	4.50	4.70
run3	4.55	4.68
run4	4.072	4.24
run5	4.19	4.37
run6	4.38	4.56
run7	4.08	4.31
run8	4.20	4.38
run9	4.33	4.52
run10	4.50	4.69
stdev	0.17	0.16
uniform	5.75	5.96
ALUs	3.86	4.02

Table 7.2: Wire-length results

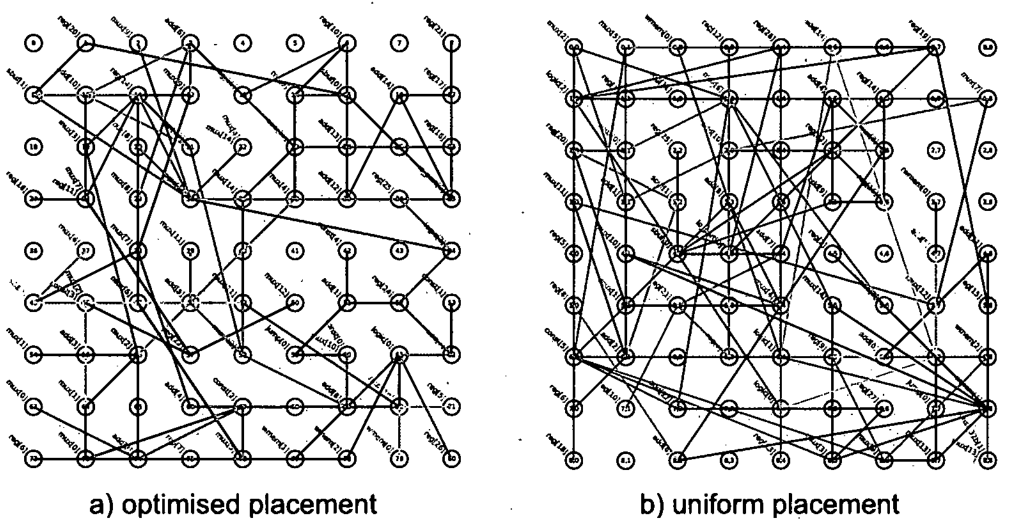


Figure 7.20: Optimised and uniform placement comparison

phase makes use of a priority based composite fitness, as described in 7.5. It has 3 objectives, of which, one is to reduce conflicts, the other to minimise the wire-lengths and the last to increase fan-out sharing, with priorities that follow that order.

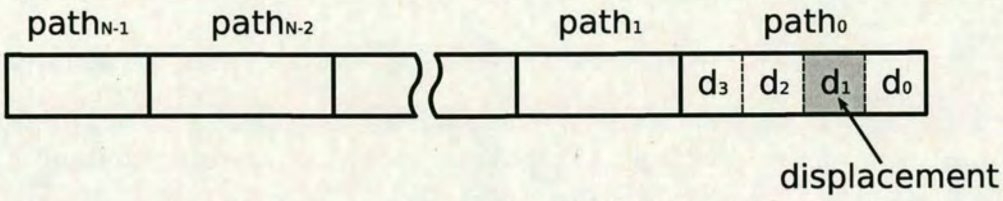


Figure 7.21: Routing chromosome encoding

The routing chromosome, shown on figure 7.21, contains those parameters that affect the path's shape and which are modifiable. Since the source-destination distance, as well as the monotonicity, are constant properties of a path, they are not encoded in the chromosome. These properties are derived from the abstract connection-pair data model of the netlist, during the decoding process and the initial calculation of the chromosome length.

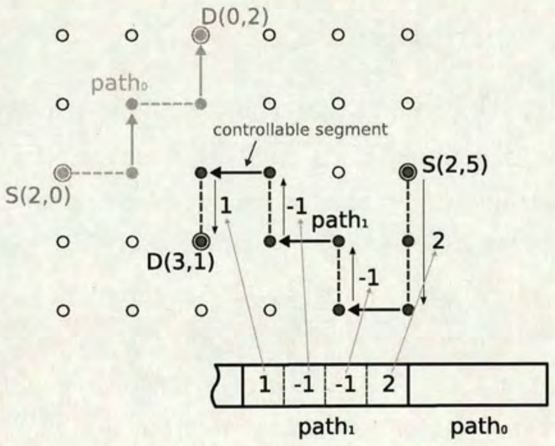


Figure 7.22: Example decoding of a -X monotonic path

The chromosome is encoded using the path-encoding technique described in section 5.3. As seen on figure 7.22, each 'controllable segment' coresponds to a gene cluster, expressing the displacement of that segment as a trait. A path is thus expressed by a series of genes describing the displacement vector of that path. The path-encoding, using the displacement format, provides a fixed size representation for each connection, which results in an easier to manage, fixed size chromosome.

A step is considered to be mapped successfully when all conflicts are eliminated. Otherwise the GA loop will continue trying until the 'idle' generation count reaches a given threshold, in which case the GA loop retires and restarts the mapping process for that step. The algorithm will try mapping the step several times, until reaching a user-defined repeat count, at which point the step is marked as 'un-routed' and the algorithm proceeds to the next. When the mapping is successful and all conflicts are eliminated, the GA loop will continue refining the tracks to further improve the secondary objectives until the 'starvation' condition occurs, at which point the algorithm emits the resulting mapping and moves to the next step.

Section 7.6.1 includes a set of routing results based on a number of placement arrangements.

7.6.3 Group Optimiser

As described in 5.13, the configuration context of the RICA prototype is partitioned into clusters or groups. Each group addresses a set of insertion points, one at a time. Multiple groups form a configuration 'row' and are loaded simultaneously. On every row, each group addresses one of its insertion points. Since each row has a fixed size, as seen in 5.17, and may not be fully populated, the configuration stream ends up being fragmented with several empty slots. The amount of fragmentation depends on the insertion points in use and how these are organised in groups. For every insertion point in the configuration context, that is part of the same group, a new row needs to be formed. The more rows a configuration context is divided to, the higher the fragmentation, as well as the loading time.

In order to maximise row utilisation and reduce the loading times, the configuration contexts are analysed and statistics are collected for the usage of all insertion points. Based on this analysis, appropriate group partitions are chosen that increase row efficiency. This process is of course application domain dependent, since it is based on the configuration context of a given application or set of applications. Nevertheless, the results show that given a good set of applications, a generic enough partitioning can be made.

The group partitioning analysis tool described here is once again based on the GA framework described in 7.5.

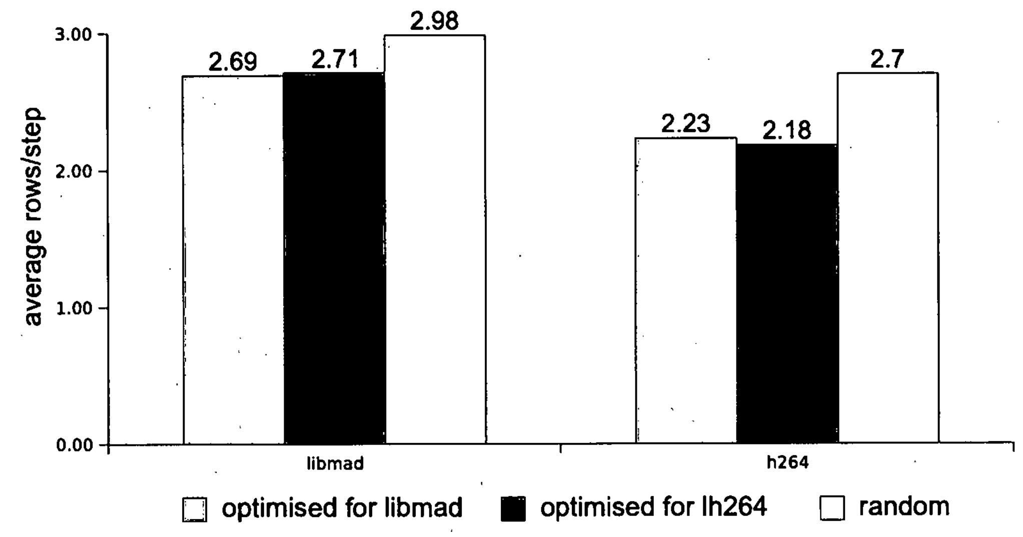


Figure 7.23: Average rows/step for MP3(libmad) and h264

7.6.4 Chain Generator

As described in 5.6, a chain is a sequence of turn symbols used to describe path-shapes. In a way it forms the 'dictionary' in an encoding that is similar to dictionary-based compression. With this encoding, a path shape becomes a starting index, to this dictionary and a path length parameter.

Given a chain, there are a finite number of possible path shapes that can be described. Every shape is a sequence of symbols from a given starting point. The 'coverage' of the chain is an important figure that defines the efficiency of that chain. Once again, an analysis of the paths used in one ore more routed netlist can provide the required information to produce a chain with high coverage.

	libmad MP3	h264 Decoder
max length	16	21
avg length	3.66	2.78
unique shapes	203	783
total count	8378	142194
%symbol coverage	97.7	95.34
%path coverage	95.5	95.5

Table 7.3: Chain encoding with 255 symbols

Table 7.3 shows the results obtained after chain encoding the netlist of libmad MP3 decoder and the h264 decoder implementation. These results show that a 255 symbol list achieves high path coverage.

Chapter 8

SMT on RICA

8.1 The SMT design paradigm

In programming environments, threads are uncorrelated, independent tasks working in 'parallel', ideally on separate sets of data. Throughout the execution of a multi-threaded(MT) program there is no pre-defined or known order in which the threads are executed or synchronous to each other. If a known pattern could be constructed from analysing the program then it would be questionable to why one should implement this as an MT program.

A multi-threaded program relies on dynamic information which is not known at compile time. Because of that it is practically impossible to statically analyse the execution flow of a multi-threaded program and make allocation decisions that will satisfy all active supported threads. Each thread is mapped independently without any prior knowledge of other active threads.

Traditionally, true multi-threading is associated with the use of completely isolated execution cores. Conventional MT on a single core can be achieved by time-slicing the hardware resources, often referred to as temporal multithreading. With the appearance of superscalar computers, true MT is possible even in a single core. Inherently parallel architectures rely on the use of a large number of computational resources and/or interconnection resources (in case of reconfigurable computing). Keeping these resources busy is a challenging task. Most parallel architectures suffer from low core utilisation. An attempt to reduce this effect is to employ Simultaneous Multi-Threading (SMT) in a single core [68, 69]. SMT usually comes as an extension to the execution flow control mechanism of a core, allowing sharing most of the cores resources by more than one concurrent thread. The realisation of SMT is very implementation dependent but a general overview for a given class of devices can be defined. One of the commercially available SMT implementation can be found in Intel's Pentium 4 generation of microprocessors [70], as well as IBM's PowerPC architectures [71].

To achieve resource sharing, SMT requires some duplication of key components of the architecture. These typically include:

- the flow control mechanism

- the architecture state

On IBM's and Intel's advanced architectures before the introduction of SMT, these resources had been already duplicated to allow out of order execution, along with duplication of the ALUs. SMT came as a realisation that, with some modification, the hardware could also be used to run multiple threads. Other than achieving higher core utilisation, SMT can hide cache latencies and fetch and decode latencies by using this time to perform useful operations. A stall condition for one thread can leave other threads running and keep the resources busy.

8.2 SMT in reconfigurable computing

RICA is used here to describe an SMT implementation on a reconfigurable computing architecture. However, the description is kept general enough to cover a wide range of architectures, as long as the following assumption are met:

The architecture should provide:

- a collection of computational units (functional and storage)
- a flexible interconnection structure to communicate information among these units
- means to control the above to form useful operations

RICA provides a number of heterogeneous computational units and a flexible 2-dimensional interconnection network allowing the formation of arbitrary complex data spathes. Flow control is performed via the 'jump' cell, which modifies the program counter conditionally or unconditionally. In the former case, the condition is given as an input to the 'jump' cell and can be driven by any cell in the array, thus allowing it to be control ed by complex logic. The 'architecture state' in RICA refers to the actual configuration context, which includes cell modes and interconnection setup.

As mentioned in section 8.1, both the flow control mechanism and the 'architecture state' need to be replicated for each SMT thread. Introducing a second 'jump' cell is something trivial. However, the fetching unit requires some modification to accommodate both 'jump' cells and more specifically the addition of an arbitration logic. Assuming that the program memory interface bandwith remains the same and to avoid duplicating the configuration stream decoder logic, the arbitration logic is placed between the fetch unit and the 'jump' cells, as seen in figure 8.1.

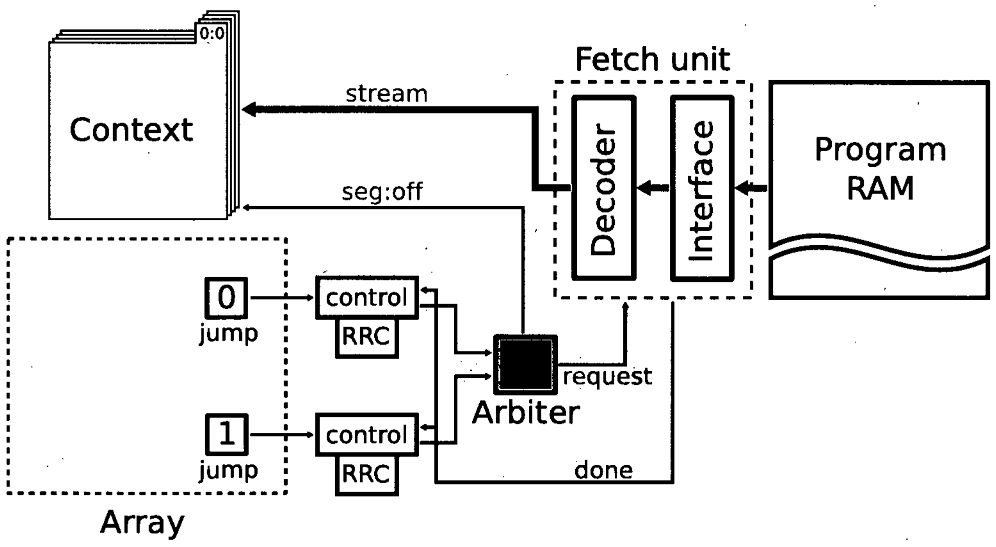


Figure 8.1: SMT: Arbitration coupling

Each SMT thread maintains a complete configuration context, which forms the 'architecture state'. The arbiter serves fetch requests from the 'jump' cells and directs the fetched configuration stream to the appropriate configuration context. The two threads share the same fetch unit and thus program interface. As seen in section 6, in a mostly kernel based environment, the program memory bandwidth requirements are very low.

Each thread is scheduled and mapped to the array resources separately. In temporal multithreading, only one thread is active at any given time. In an SMT environment, these threads will occupy different parts of the array and could operate in parallel. The example on Figure 8.2 shows 2 threads that share the same reconfigurable array without any conflicts. This is an ideal case which maximises the array utilisation. Attempts to enforce such thread isolation can be made at compile time by restricting one of the threads to one corner of the core while allocating the other one to the opposite side of the core.

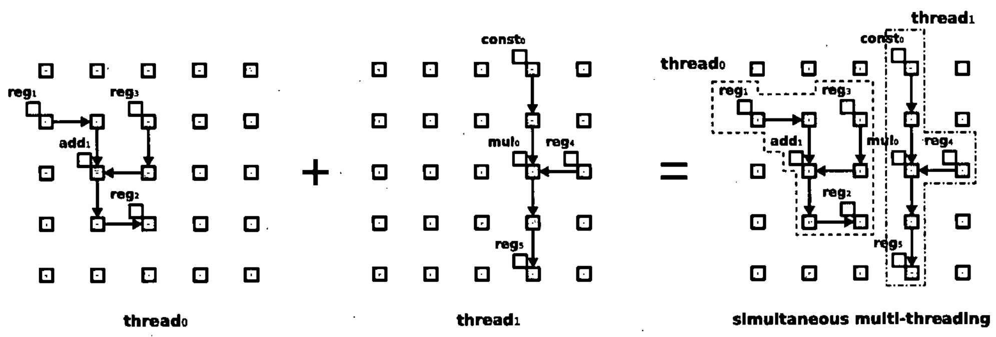


Figure 8.2: SMT: Non-Conflicting Threads

In most cases, 2 concurrent threads would request one or more common resource at the same time. A form of interconnection resource 'conflict' is shown on Figure 8.3. In this example, the common resource will be assigned to one of the two threads first, while the other one would need to wait.

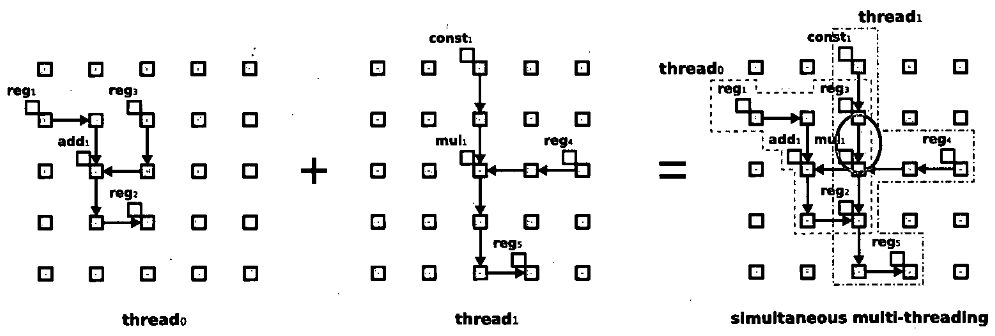


Figure 8.3: SMT: Conflicting Threads

8.3 Performance Improvements

A typical thread might have more than one independent data-path in a given configuration. Unless a strict synchronisation is imposed, each independent non-conflicting data-path can operate in parallel to the rest. This could result in a thread being executed partially, i.e. only some of its data-paths. The use of partial execution can further increase core utilisation and as a consequence, increase the performance. Figure 8.4 shows the execution time chart for two threads running under various scenarios, such as temporal MT, SMT and SMT with support for partial executions.

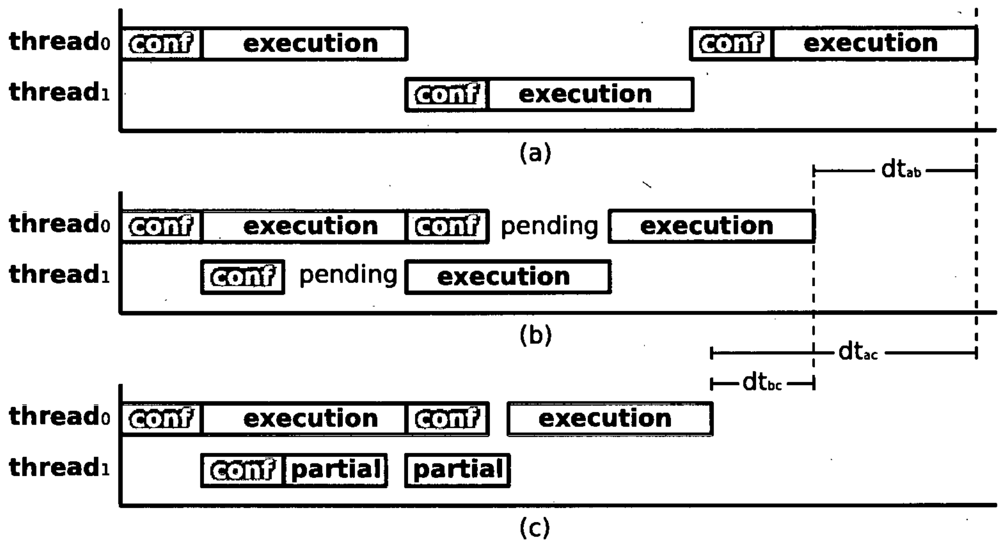


Figure 8.4: Execution of two threads using (a) Temporal multithreading, (b) SMT, (c) SMT supporting partial execution

Chapter 9

Conclusion

9.1 Contribution

This thesis makes evolutionary steps in the development of a high performance programmable reconfigurable devices and data-flow machines in general. The outcome of this work is delivered as an architecture template, with an abstract representation allowing one to deploy arrays with different resource mix and layout setups as well as accompanied software tool-flow. The latter is made architecture setup agnostic by the use of a common representation/language.

There are a number of enabling factors that have been described here, which help to achieve high performance figures in terms of speed, area and power, while maintaining a programming model comparable to GPPs. The use of a single autonomous fabric, for instance, capable of handing both flow control and computational data-paths, maintains a simple programming model comparable to GPPs'. The use of heterogeneous functional cells allows one to tailor the architecture to a given domain and save area in the process. The distributed registers enable parallel access and reduced power that is otherwise associated with accessing a register file.

Bigger emphasis was given to the reconfiguration subsystem, configuration distribution and interconnects, as well as placement optimisations. This work has demonstrated that a carefully chosen physical placement can reduce the negative effects of a heterogeneous design on the routing side. With help of meta-heuristics, based on the connection patterns obtained by analysing a set of algorithms of a given domain, an optimised heterogeneous layout was produced with comparable performance to a homogeneous layout, in terms of routability metrics.

To reduce the configuration times and program memory footprint, an innovative code compression technique has been developed that matches the intrinsic characteristics of a reconfigurable fabric. Based on a connection-oriented approach and a distributed dictionary based compression, this novel code compression technique has managed to reduce program memory activity by 2.75 times on the tested multimedia applications.

Finally, this work also takes the first steps towards the implementation of a SMT extension for reconfigurable cores and demonstrates the potential for a partial execution technique, which offers further performance improvements.

9.2 Future work

Reconfigurable architectures, such as RICA, are still young with a plethora of exciting ideas yet to be explored or ported over to this new design paradigm. This section list some of those ideas that could benefit the development of RICA and other reconfigurable architectures in general.

9.2.1 Thread partitioning

Similar to the SMT approach described in 8, the core requires a control flow unit and the associated RRC, per supported thread. The configuration context is partitioned into slices that correspond to different sections of the core. Each slice is assigned to one flow control unit. However, a master flow control unit can also take control of one or more of the other slices. In that way, several partitioning scenarios are possible. The partitioning is stored in the configuration context, alongside the rest of the configuration.

One of the most challenging parts is the memory access. Each thread operates on its own time domain, which results in complex access patterns. One solution is to have separate physical memories for each 'slice'. Some of these issues are addressed in 8.2. Another aspect that needs attention, is thread synchronisation. If dynamic re-partitioning is allowed, which is desirable, it makes thread synchronisation even more complex.

9.2.2 Multi-Core

Multi-core is the natural progression for a programmable architecture. A synergy of programmable cores combining the power of each individual core to increase parallelism and thus achieve higher performance. Unlike thread partitioning in section 9.2.1, each core is stand alone with its own reconfigurable fabric, memory interfaces and control flow. The available coupling scenarios are similar to those mentioned in 2.4. In addition, a shared memory coupling can be done on a local common cache level, thus hiding some of the latencies going off-chip. As for load balancing and responsibilities, the cores can work as independent peers, conforming to a decentralised model or a more conventional master-slave approach, where one of the cores becomes the coordinator.

Research in multi-core systems borrows a lot from multi-processor and has been studied extensively in the literature. Most of the things that apply to conventional μP are valid for RPUs as well, and thus the basic principles are similar to those found in multi-core architectures such as picoChip[19], Ambric[17] and RAW[20]. Basic features such as 'atomic' operations when accessing shared resource, allow the development of thread synchronisation protocols and thus enable a safe multi-threading environment.

Lightweight NoC, such as that proposed in section 4.3.1 can provide the basis of a decentralised multi-core platform.

9.2.3 Self-timed

RICA is 'told' how long each context should persist for before switching to the next, or iterate the current one. This information, which is in effect the critical path delay time used by the RRC unit, is stored in the configuration context and is pre-calculated during the compilation face. The calculation is based on extracted characteristics from the core and tends to be fairly accurate. However, the core uses a finite time resolution, which is defined by its master clock. The resulting quantisation error can be as high as close to 2x the target critical path delay, when the timing resolution is comparable to the critical path.

Furthermore, in some cases, a configuration context may include more than one conditional path-ways that are highly unbalanced. In such cases, the resulting run-time critical path is data driven and can vary significantly. The RRC expiration time is static. There are cases where the largest critical path-way occurs only a fraction of the time, thus wasting idle cycles.

Both of the above cases can impact performance by reducing the computational density of the core. A possible solution is to have a self-timed core. A self-timed core is able to detect when calculations have finished and thus issue an appropriate event for reconfiguration or next iteration. Self-timing is usually done using asynchronous techniques as mentioned in 6.2.

9.2.4 Dynamic Allocation and Routing

The RICA core can dynamically reconfigure itself and form new data-paths. However, those data-paths are mapped during compile time and stored in the configuration context. That is, the cell instances used and the paths formed are pre-defined. Both allocation and routing are NP-complete problems and are normally tackled with heuristic and meta-heuristic algorithms. The implementation complexity of a class of such algorithms is shown in 7.5. Due to that complexity, the mapping process, which involves allocation and routing, is done as a pre-processing stage and the result is stored in the core's program memory.

Most application scenarios are well served by the above approach. However, there are some usage models that can benefit from, or even require, dynamic allocation and routing. One such example is the SMT extention shown in 8. Normally a thread would freeze if some routing or computation resource it requires is occupied. With dynamic mapping, the threads could more efficiently negotiate the resources, thus reducing, if possible, the wait time. In a similar way, the 'static' thread partitioning, discussed in 9.2.1, would benefit from dynamic mapping. In fact doing so would make it similar to the SMT approach, where threads negotiate resources.

Fault tolerance is another feature that would benefit from dynamic mapping. Traditionally, fault tolerance can be achieved by coupling the core with an external host processor that is able to run the mapping algorithms and update the configuration context accordingly. Having a built in dynamic mapping would simplify and speeds up this process. Once a fault is detected the core is able to re-evaluate its mapping and bypass it.

A large part of the required structures is already present in the core, such as the spatial relationship between the various resource. There are numerous techniques in literature for fault-tolerance in networks [53] and other adaptive techniques that could apply in RICA. For instance, an implementation of the

Dijkstra shortest path algorithm [72], in a distributed fashion on the array could offer an interesting approach to dynamic routing.

Dynamic allocation and routing would most likely require a self-timed RICA, as mentioned in section 9.2.3.

9.2.5 Extended vector arithmetic

As discussed in 3.6, the efficient use of SIMD instructions requires the existence of convenient unpacking operators. The RICA prototype makes use of special modes added to the Logic cell, which are found to limit the extent in which SIMD instructions can be used. A more flexible solution would be to enable unpacking and shuffling operators on the interconnection level. Define a small set of simple unpacking operators, which if combined in series can form more complex modifiers. Each of the operators should be small enough to avoid increasing the overall area of the design. With such modifiers in place, complex data-flow graphs utilising SIMD instruction will not impact routing as much and will not be limited by the otherwise small number of unpacking operators.

9.2.6 Source and Target graph matching

As seen in section 7.6, the mapping process of a source data-flow graph to the target architecture, or 'target graph', is a complex, often multi-objective problem. Given a source graph the mapping algorithms will try to choose an allocation that will offload some of the routing complexity, by minimising wire-length and shifting potential conflicts, where there are more free resources. Nonetheless, the end result is highly dependent on the input source graph and how that compares to the target interconnection architecture.

Empirical results obtained by E.F Rent by observing several designs in terms of pin count and gate count, led to the formulation of a relationship between those features, which is expressed by the $N_p = K_p \times N_g^{\beta}$ equation, known as Rent's rule [73]; where N_p is the number of pins, N_g is the number of gates and K_p and β are the proportionality constant and Rent's constant respectively. The Rent's rule predicts an exponential increase in interconnection complexity as the number of elements increase and as such has been used to estimate wire-lengths and overall interconnection complexity of designs[74, 75]. By dividing a design into several partitions, each of those partitions is subject to Rent's rule. Such a dissection of a design can reveal intrinsic properties for the interconnects of the design, which can be useful when mapping it onto a target architecture. More specifically, the source graph of a design can be analysed and its suitability for a given target architecture/graph measured. One can use such a classification to derive a suitable interconnection architecture for the target platform, based on the source data-flow graphs the architecture is subjected to. In addition, once an architecture is chosen, the source graph analysis can reveal potential problems that will appear during mapping of the source graph to the fixed target architecture. One can then modify the source graph accordingly to conform to certain characteristics, which would make it more appropriate for the target architecture.

Source-target graph matching can offer an extra help in mapping a given design to the target reconfigurable hardware and thus improve the design traits, such as speed and computational density. It could also be the differentiating factor between a successful mapping and one that fails due to global or local routing resource starvation.

Glossary

μP	micro Processor, 14, 16, 29, 156
ALU	Arithmetic Logic Unit, 21, 29, 46, 139
ASIC	Aplication Specific Integrated Circuit, 1, 14, 16–18, 33, 36, 56, 108
ASIP	Application Specific Instruction Processors, 31
CFG	Control Flow Graph, 45
CLB	Configurable Logic Block, 28, 29, 37, 56
CPU	Central Processing Unit, 30–32, 34, 45
DAC	Digital to Analogue Converter, 41
DMA	Direct Memory Access, 41, 46
DSP	Digital Signal Processor, 14, 16, 18, 33, 34
EA	Evolutionary Algorithm, 37
FE	Functional Element, 25
FPGA	Field Programmable Gate Array, 14, 16, 17, 24, 25, 27–30, 34, 35, 37, 47, 54, 56, 59
FSM	Finite State Machine, 120
GA	Genetic Algorithm, 37, 145

GPP	General Purpose Processor, 1, 14, 17–20, 24, 31, 33, 42, 44, 108, 154
HDL	Hardware Description Language, 16, 33, 36, 119, 121
HEM	Hierarchical Enable Matrix, 8, 82, 86, 95, 96
HPC	High Performance Computing, 24
ID	Identification, 122
ILP	Instruction Level Parallelism, 16, 45
ISA	Instruction Set Architecture, 113
LSB	Least Significant Bit, 84
LUT	Look Up Table, 29, 78
MDF	Machine Description File, 109
MIPS	Million Instructions Per Second, 31
MP	Multi-Processor, 30, 125
MSB	Most Significant Bit, 84
MT	Multi-Threading, 125, 148, 151
NoC	Network on Chip, 29, 59, 156
NP	Non-deterministic Polynomial time, 37, 157
NRE	Non-Recurring Engineering, 16, 17, 25, 33
PLL	Phase Lock Loop, 53
PnR	Place and Route, 37
RAM	Random Access Memory, 94
RC	Reconfigurable Computing, 19, 22, 32, 36, 57
RCD	Reconfigurable Computing Device, 25, 28

RICA	Reconfigurable Instruction Cell Array/Architecture, 1, 18–20, 38, 40, 42, 44, 45, 47–49, 53, 54, 57, 75, 77, 87, 100, 103–105, 108–111, 117, 155, 158
RISC	Reduced Instruction Set Computer, 1, 20, 29, 30, 34, 44, 99
RLC	Run Length Coding, 35
RPU	Reconfigurable Processing Unit, 30–32, 34, 156
RRC	Reconfiguration Rate Controller, 52, 100, 101, 119, 120, 155, 156
SA	Simulated Annealing, 37
SDR	Software Defined Radio, 31
SIMD	Single Instruction Multiple Data, 1, 19, 28, 31, 50, 158
SLIg	System Level Integration group, 18, 19
SMT	Simultaneous Multi-Threading, 38, 59, 148–151, 154, 155, 157
TLM	Transaction Level Model, 118, 119
VCDRC	Virtual Channels using Dynamic Rate Control, 61, 62, 68
VLIW	Very Long Instruction Word, 14, 16, 29, 34, 35, 91

Bibliography

[1] S. Khawam, I. Nousias, M. Milward, Y. Yi, and T. Arslan, "Reconfigurable instruction cell array," International Patent Office, Tech. Rep., 2006.

[2] I. Nousias, S. Khawam, M. Milward, M. Muir, Y. Yi, and T. Arslan, "Simultaneous multi-threading on rica," International Patent Office, Tech. Rep., 2006.

[3] ——, "Encoding and decoding methods," International Patent Office, Tech. Rep., 2008.

[4] S. Khawam, I. Nousias, M. Milward, M. Muir, Y. Yi, and T. Arslan, "The reconfigurable instruction cell array," *IEEE Trans. VLSI Syst. 16*, 2008.

[5] S. Khawam, S. Baloch, A. Pai, I. Ahmet, N. Aydin, T. Arslan, and F. Westall, "Efficient implementations of mobile video computations on domain-specific reconfigurable arrays," in *Desogn Automation and Test in Europe*, vol. 2, New York, February 2004, pp. 1230–135.

[6] S. Khawam, "Reconfigurable architectures for low-power SoC: Domain-specific and RICA based systems," Ph.D. dissertation, The University of Edinburgh, April 2006.

[7] Y. Yi, I. Nousias, M. Milward, S. Khawam, T. Arslan, and I. Lindsay, "System-level scheduling on instruction cell based reconfigurable systems," in *DATE'06*.

[8] G. Estrin and C. R. Viswanathan, "Organization of a "fixed-plus-variable" structure computer for computation of eigenvalues and eigenvectors of real symmetric matrices," in *In Proceedings of the Western Joint Computer Conference*, New York, 1960, pp. 33–40.

[9] H. Singh, M. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Filho, "Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications," in *IEEE Transactions on Computers*, vol. 49, no. 5, 2000, pp. 465–481.

[10] J. Hauser and J. Wawrzynek, "GARP: A mips processor with a reconigurable coprocessor," in *IEEE Symposium on FPCCM*, 1997.

[11] "D-Fabrix processing array, reconfigurable signal processor," 2005. [Online]. Available: www.elixent.com

[12] PACT, "OFDM decoder for wireless LAN - whitepaper," May 2002. [Online]. Available: www.pactcorp.com

[13] J. Osburn, W. Anderson, R. Rosenberg, and M. Lanzagorta, "Early experiences on the nrl cray xd1," in *HPCMP-UGC '06: Proceedings of the HPCMP Users Group Conference.* Washington, DC, USA: IEEE Computer Society, 2006, pp. 347–353.

[14] R. Usselmann, "Soc bus review," OpenCores, Tech. Rep.

[15] OpenCores, "Wishbone system-on-chip (soc) interconnection architecture for portable ip cores," OpenCores, Tech. Rep.

[16] C. Ebeling, C. Fisher, G. Xing, M. Shen, and H. Lui, "Implementing an OFDM receiver on the rapid reconfigurable architecture," in *IEEE Trans. Computers*, vol. 53, November 2004, pp. 1436–1448.

[17] Abric, "Ambric programmable multicore," 2007. [Online]. Available: www.ambric.com

[18] Philips, "Avispa: Silicon Hive," 2005. [Online]. Available: www.siliconhive.com

[19] R. Baines and D. Pulley, "A total cost approach to evaluating different reconfigurable architecture for baseband processing in wireless receivers," in *IEEE Trans. Computers*, vol. 41, January 2003, pp. 105–113.

[20] M. Taylor, "The raw prototype design document," in *Department of Electrical Engineering and Computer Science*, October 2003.

[21] P. Master, "The next big leap in reconfigurable systems," in *Field-Programmable Technology 200*, December 2002, pp. 17–22.

[22] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix." in *FPL*, vol. 2778. Springer, 2003, pp. 61–70.

[23] K. Sayano and T. Shirakawa, "Pleiades: A prototype of inter-processor network generation system," in *ISPAN '97: Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '97).* Washington, DC, USA: IEEE Computer Society, 1997, p. 202.

[24] Stretch, "Strech configurable processor," 2007. [Online]. Available: www.stretchinc.com

[25] T. Vogt and N. Wehn, "A reconfigurable asip for convolutional and turbo decoding in an sdr environment," in *VLSI Systems'08*, vol. 16, 2008, pp. 1309–1320.

[26] S. Saponara, M. Casula, and L. Fanucci, "Asip-based reconfigurable architectures for power-efficient and real-time image/video," in *Real-Time Image Processing'08*, vol. 3, no. 3, September 2008, pp. 201–216.

[27] ustin L. Tripp, K. D. Peterson, C. Ahrens, and J. D. Poznanovic, "An fpga compiler framework for floating-point algorithms," in *Field Programmable Logic and Applications*, 2005, pp. 317–322.

[28] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded risc architecture," in *Proc. of the Intl. Symposium on Microarchitecture*, December 1992, pp. 81–91.

[29] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression for vliw processors using variable-to-fixed coding," in *ISSS '02: Proceedings of the 15th international symposium on System Synthesis.* ACM, 2002, pp. 138–143.

[30] G. Hansel, U. D. Rouen, D. Perrin, U. D. Paris, and I. Simon, "Compression and entropy," in *In Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science.* Springer-Verlag, 1992, pp. 515–528.

[31] N. J. Larsson and A. Moffat, "Offline dictionary-based compression," in *Proc. IEEE.* IEEE Computer Society, 1999, pp. 296–305.

[32] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression for vliw processors using variable-to-fixed coding," in *ISSS '02: Proceedings of the 15th international symposium on System Synthesis.* ACM, 2002, pp. 138–143.

[33] B. Gorjiara, M. Reshadi, and D. Gajski, "Merged dictionary code compression for fpga implementation of custom microcoded pes," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 2, pp. 1–21, 2008.

[34] S. Kirkpatrick, C. G. Jr., and M. Vecchi, "Optimization by simulated annealing," in *Science*, 2000, pp. 671–680.

[35] T. Blickle and L. Thiele, "A comparison of selection schemes used in genetic algorithms," Gloriastrasse 35, 8092 Zurich, Switzerland, Tech. Rep. 11, 1995.

[36] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Design Automation Conference*, 1982, pp. 175–181.

[37] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Field Programmable Logic*, 1997.

[38] B. C. Kahne, "A genetic algorithm-based place-and-route compiler for a run-time reconfigurable computing system," Ph.D. dissertation, Virginia Polytechnic Institute and State University, 1997.

[39] J. Hennessy, D. A. Patterson, D. Goldeberg, and K. Asanovic, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2003, ch. Historical Perspective and References.

[40] D. A. Patterson and D. R. Ditzel, "The case for the reduced instruction set computer," vol. 8, October 1980, pp. 25–33.

[41] D. W. Wall, "Limits of instruction-level parallelism," 1991, pp. 176–188.

[42] J. Handy, *The cache memory book.* San Diego, CA, USA: Academic Press Professional, Inc., 1993.

[43] D. J. Lilja, "Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons," *ACM Comput. Surv.*, vol. 25, no. 3, pp. 303–338, 1993.

[44] J. Stewart and G. Simmons, "An investigation of simd instruction sets," Tech. Rep., 2005.

[45] G. Varghese, "Low energy field-programmable gate array," Ph.D. dissertation, University of California, Berkeley, 2000.

[46] E. S. Ochotta, P. J. Crotty, C. R. Erickson, C.-T. Huang, R. Jayaraman, R. C. Li, J. D. Linoff, L. Ngo, H. V. Nguyen, K. M. Pierce, D. P. Wieland, J. Zhuang, and S. S. Nance, "A novel predictable segmented fpga routing architecture," in *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays.* New York, NY, USA: ACM, 1998, pp. 3–11.

[47] G. Lemieux, E. Lee, M. Tom, and A. Yu, "Directional and single-driver wires in fpga interconnect," in *Field Programmable Technology*, 2004, pp. 41–48.

[48] L. M. Ni and P. K. McKinley, "A survey of wormhole routing techniques in direct networks," *Computer*, vol. 26, no. 2, pp. 62–76, 1993.

[49] R. M. Jingcao Hu, "DyAD smart routing for networks-on-chip," in *Design Automation Conference*, June 2004.

[50] K. M. Al-Tawil, M. Abd-El-Barr, and F. Ashraf, "A survey and comparison of wormhole routing techniques in a meshnetworks," vol. 11, April 1997, pp. 38–45.

[51] Z. Lu and A. Jantsch, "Flit ejection in on-chip wormhole-switched networks with virtual channels," 2004.

[52] S. Khorandy and A. Leon-Garcia, "Robust non-probabilistic bounds for delay and throughput in credit-based flow control," in *INFOCOM*, 1996, pp. 577–584.

[53] A. Pullini, F. Angiolini, D. Bertozzi, and L. Benini, "Fault tolerance overhead in network-on-chip flow control schemes," in *SBCCI '05: Proceedings of the 18th annual symposium on Integrated circuits and system design.* New York, NY, USA: ACM Press, 2005, pp. 224–229.

[54] "Direct network overview."

[55] A. Davis and S. M. Nowick, "An introduction to asynchronous circuit design," The Encyclopedia of Computer Science and Technology, Tech. Rep., 1997.

[56] A. Bink and R. York, "Arm996hs: The first licensable, clockless 32-bit processor core," in *IEEE Micro '07*, vol. 27, April 2007, pp. 58–68.

[57] "Handshake solutions." [Online]. Available: www.handshakesolutions.com

[58] "Gnu compiler collection." [Online]. Available: http://gcc.gnu.org/

[59] D. C. Black and J. Donovan, *SystemC: From The Ground Up.* Kluwer Academic, 2004.

[60] "Open systemc initiative." [Online]. Available: www.systemc.org

[61] F. Mueller, "A library implementation of POSIX threads under Unix," in *Proceedings of the Winter 1993 USENIX Technical Conference and Exhibition*, San Diego, CA, USA, 1993, pp. 29–41.

[62] J. Merelo, M. Keijzer, and M. Schoenauer, "Eo evolutionary computation framework." [Online]. Available: http://geneura.ugr.es/ jmerelo/EO.html

[63] M. Faupel, "Gajit: A simple java genetic algorithms package." [Online]. Available: http://www.micropraxis.com/gajit/index.html

[64] CoolSoft, "Genetic algorithm library," 2008. [Online]. Available: http://www.coolsoft-sd.com/Products.aspx?pid=1

[65] D. Levine, "Pgapack: Parallel genetic algorithm library," 2000. [Online]. Available: http://www-fp.mcs.anl.gov/CCST/.../pgapack.html

[66] M. Wall, "Galib: A c++ library of genetic algorithm components," 1996. [Online]. Available: http://lancet.mit.edu/ga/

[67] S. Adcock, "Gaul: Genetic algorithm utility library," 2000. [Online]. Available: http://gaul.sourceforge.net/

[68] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *In 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 392–403.

[69] D. Susan and H. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," June 1995.

[70] D. Koufaty and D. T. Marr, "Hyperthreading technology in the netburst microarchitecture," in *IEEE Micro '03*, 2003, pp. 56–65.

[71] H. Mathis, A. Mericas, J. McCalpin, R. Eickemeyer, and S. Kunkel, "Charactization of the simultaneous multithreading(smt) efficiency in power5," IBM, Tech. Rep., September 2005.

[72] E. W. Dijkstra, "A note on two problems in connexion with graphs," in *In Numerische Mathematik*, January 1959, pp. 269–271.

[73] H. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI.* Addison Wesley, 1990, ch. Machine Organization and Rent's Rule, pp. 416–421.

```
        input_port = SOUTH;
        y--;
      }
    }
  }

  if(monotonic==0)
  {
    if(direction==0)
    {
      s(x,y).south = input_port;
      input_port = NORTH;
      y++;
    }
    else
    {
      s(x,y).north = input_port;
      input_port = SOUTH;
      y--;
    }
  }
  else
  {
    if(direction==0)
    {
      s(x,y).east = input_port;
      input_port = WEST;
      x++;
    }
    else
    {
      s(x,y).west = input_port;
      input_port = EAST;
      x--;
    }
  }
}

s(x,y).terminal = input_port;
```

Appendix B

RICA Netlist Formal Language Syntax

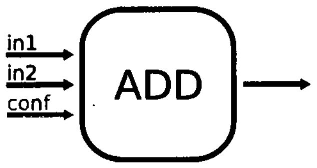
```
<source_text>
    ||= <directive>
    ::= <description_field>

<directive>
    ::= <define_directive>*
    ||= <include_directive>*

<define_directive>
    ::= 'define <define_name> <replacement>

<define_name>
    ::= <IDENTIFIER>

<replacement>
    ::= <IDENTIFIER>
    ::= <formatted_number>

<include_directive>
    ::= 'include " <netlist_file> "
    ||= 'includev " <verilog_file> "

<netlist_file>
    ::= <IDENTIFIER>

<verilog_file>
    ::= <IDENTIFIER>

<description_field>
    ::= <architecture>+
    ||= <data>+
    ||= <sequence>+
<architecture>
    ::= architecture { <cell>+ <layout>+ }
    ||= architecture <cell>
    ||= architecture <layout>
    ||= architecture <row>

<cell>
    ::= cell { <cell_declaration>* }
    ||= cell <cell_declaration>

<cell_declaration>
    ::= <cell_name> { <port_declaration>* }
    ||= <cell_name> <port_declaration>

<cell_name>
    ::= <IDENTIFIER>

<port_declaration>
    ::= <port_type> <port_definition><,<port_definition> ;

<port_definition>
    ::= <port_name> <width_definition>?

<port_type>
    ::= output
    ||= input
    ||= config

<port_name>
    ::= <IDENTIFIER>

<width_definition>
    ::= [ <PLAIN_NUMBER> ]

<layout>
    ::= layout { <cell_setting>* }
    ||= layout <cell_setting>

<cell_setting>
    ::= . <cell_name> <port_setting>
    ||= . <cell_name> { <port_setting>* }

<cell_index>
    ::= <PLAIN_NUMBER>

<port_setting>
    ::= . <port_name> <port_parameter_assign>
    ||= . <port_name> { <port_parameter_assign>* }
```

```
<port_parameter_assign>
    ::= . <port_parameter> = <PLAIN_NUMBER> ;

<port_parameter>
    ::= y
    ||= x
    ||= p

<row>
    ::= <row_definition>
    ||= { <row_definition>* }

<row_definition>
    :: = ( <PLAIN_NUMBER> ) . <cell_name> ;

<data>
    ::= data { <storage>+ }
    ||= data <storage>

<storage>
    ::= ram { <storage_element>+ }
    ||= ram <storage_element>
    ||= rom { <storage_element>+ }
    ||= rom <storage_element>

<storage_element>
    ::= . align <PLAIN_NUMBER> ;
    ||= . space <PLAIN_NUMBER> ;
    ||= . byte <PLAIN_NUMBER> ;
    ||= . long <PLAIN_NUMBER> ;
    ||= . word <storage_entry>

<storage_entry>
    ::= <PLAIN_NUMBER> ;
    ||= <formatted_number> ;
    ||= <step_link> ;
    ||= <data_link> ;

<data_link>
    ::= ! <data_rom_label>
    ||= # <data_ram_label>

<data_rom_label>
    ::= <IDENTIFIER> ;

<data_ram_label>
    ::= <IDENTIFIER> ;

<sequence>
    ::= sequence { <step>+ }
    ||= sequence <step>

<step>
    ::= step [ <step_index> ] { <cell_configuration>+ }
    ||= step [ <step_index> ] <cell_configuration>

<step_index>
    ::= <PLAIN_NUMBER>
    ||= % <step_label>

<step_label>
    ::= <IDENTIFIER>

<cell_configuration>
    ::= . <cell_name> [ <cell_index> ] { <port_assignment>* }
    ||= . <cell_name> [ <cell_index> ] <port_assignment>

<cell_index>
    ::= <PLAIN_NUMBER>

<port_assignment>
    ::= . <port_name> = <right_assignment> ;

<right_assignment>
    ::= <cell_name> . <output_port>
    ||= <formatted_number>
    ||= <PLAIN_NUMBER>
    ||= <wire>
    ||= <step_link>

<formatted_number>
    ::= <sign_symbol>? ' <radix_symbol> <PLAIN_NUMBER>

<sign_symbol>
    ::= -
    ||= +

<radix_symbol>
    ::= d
    ||= b
    ||= h

<wire>
    ::= $ <wire_name>

<wire_name>
    ::= <IDENTIFIER>

<step_link>
    ::= <absolute_address>
    ||= <relative_address>

<absolute_address>
    ::= & <step_label>

<relative_address>
    ::= @ <step_label>

<PLAIN_NUMBER>
    normal integer number

<IDENTIFIER>
    any alpharithmetic that does not start with a number. also
    character '_' is allowed
```

Appendix C

RICA Instruction Set

C.1 Adder

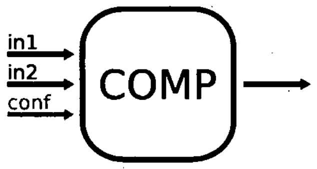


operation type	out = in1 $\pm$ in2
conf bitwidth	4
SIMD	yes
arity	binary

configuration

0 0 0 0	Single Integer	
0 0 0 1	Vector 2x Half Integer	Addition
0 0 1 0	Vector 4x Quarter Integer	
0 0 1 1	Single Integer	
0 1 0 0	Vector 2x Half Integer	Subtraction
0 1 0 1	Vector 4x Quarter Integer	
0 1 1 0	Half Integer	Complex Addition
0 1 1 1	Half Integer	Complex Subtraction
1 0 0 0	Single Integer	
1 0 0 1	Vector 2x Half Integer	Absolute Difference
1 0 1 0	Vector 4x Quarter Integer	
1 0 1 1	Single Integer	
1 1 0 0	Vector 2x Half Integer	Average
1 1 0 1	Vector 4x Quarter Integer	

C.2 Comparator

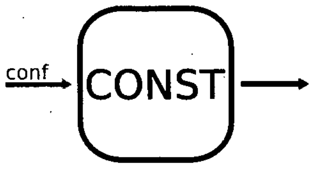


operation type	out = in1 COMP in2
conf bitwidth	4
SIMD	yes
arity	binary

configuration

0 0 0 0	Single Integer	Addition
0 0 0 1	Vector 2x Half Integer	
0 0 1 0	Vector 4x Quarter Integer	
0 0 1 1	Single Integer	Subtraction
0 1 0 0	Vector 2x Half Integer	
0 1 0 1	Vector 4x Quarter Integer	
0 1 1 0	Half Integer	Complex Addition
0 1 1 1	Half Integer	Complex Subtraction
1 0 0 0	Single Integer	Absolute Difference
1 0 0 1	Vector 2x Half Integer	
1 0 1 0	Vector 4x Quarter Integer	
1 0 1 1	Single Integer	Average
1 1 0 0	Vector 2x Half Integer	
1 1 0 1	Vector 4x Quarter Integer	

C.3 Constant

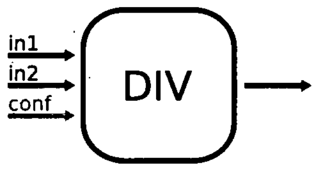


operation type	out = conf
conf bitwidth	32
SIMD	N/A
arity	nullary

configuration

x x . . . x	Constant Single Integer

C.4 Divider

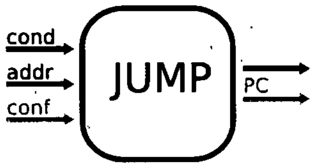


operation type	out = in1 ÷/% in2
conf bitwidth	2
SIMD	no
arity	binary

configuration

0 0	Signed Single Integer	Integer Divide
0 1	Unsigned Single Integer	Integer Divide
1 0	Signed Single Integer	Modulo
1 1	Unsigned Single Integer	Modulo

C.5 Jump

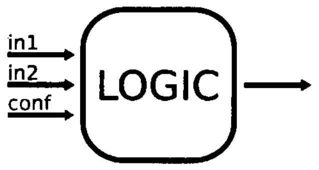


operation type	PC = cond ? addr : next=PC+1
conf bitwidth	4
SIMD	N/A
arity	binary

configuration

0 0 0 0		Always	Relative Jump
0 0 0 1		N/A	
0 0 1 0		Always	Absolute Jump
0 0 1 1		N/A	
0 1 0 0	Equal Zero	Condition Low	Relative Jump
0 1 0 1	Equal Zero	Condition High	Relative Jump
0 1 1 0	Equal Zero	Condition Low	Absolute Jump
0 1 1 1	Equal Zero	Condition High	Absolute Jump
1 0 0 0	Greater Than Zero	Condition Low	Relative Jump
1 0 0 1	Greater Than Zero	Condition High	Relative Jump
1 0 1 0	Greater Than Zero	Condition Low	Absolute Jump
1 0 1 1	Greater Than Zero	Condition High	Absolute Jump
1 1 0 0	Greater Equal Zero	Condition Low	Relative Jump
1 1 0 1	Greater Equal Zero	Condition High	Relative Jump
1 1 1 0	Greater Equal Zero	Condition Low	Absolute Jump
1 1 1 1	Greater Equal Zero	Condition High	Absolute Jump

C.6 Logic

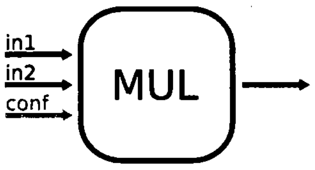


operation type	out = in1 OP in2
conf bitwidth	5
SIMD	yes
arity	binary

configuration

0 0 0 0 0	N/A	
0 0 0 0 1	Single Integer	Negate
0 0 0 1 0	Single Integer	OR
0 0 0 1 1	Single Integer	AND
0 0 1 0 0	Single Integer	XOR
0 0 1 0 1	Single Integer	NOT
0 0 1 1 0	Single Integer	NOR
0 0 1 1 1	Single Integer	NAND
0 1 0 0 0	Single Integer	Absolute
0 1 0 0 1	Single Integer	FFS
0 1 0 1 0	Single Integer	CLZ
0 1 0 1 1	Single Integer	CTZ
0 1 1 0 0	Single Integer	POPCOUNT
0 1 1 0 1	Single Integer	Parity
0 1 1 1 0	Half to Single Integer	Sign Extend
0 1 1 1 1	Quarter to Single Integer	Sign Extend
1 0 0 0 0	Quarter to 2x Half Integer	Sign Extend
1 0 0 0 1	Vector 2x Half Integer	Negate
1 0 0 1 0	Vector 4x Quarter Integer	Negate
1 0 0 1 1	Half Integer	Merge MSB
1 0 1 0 0	Half Integer	Merge LSB & MSB
1 0 1 0 1	Half Integer	Merge LSB
1 0 1 1 0	Quarter Integer	Merge Interleaved
1 0 1 1 1	Quarter Integer	Re-Order
1 1 0 0 0	Half to Single Integer	Zero Extend
1 1 0 0 1	Quarter to Single Integer	Zero Extend
1 0 1 0 0	Half Integer	Merge MSB & LSB
1 1 0 1 1	Single Integer	Bit Reverse

C.7 Multiplier

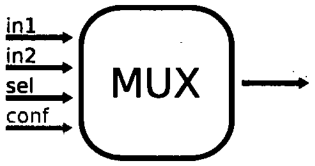


operation type	out = in1 × in2
conf bitwidth	4
SIMD	yes
arity	binary

configuration

0 0 0 0	Signed Single Integer	Low Part
0 0 0 1	Unsigned Single Integer	Low Part
0 0 1 0	Signed Double Integer	High Part
0 0 1 1	Unsigned Double Integer	High Part
0 1 0 0	Signed Vector 2x Half Integer	
0 1 0 1	Unsigned Vector 2x Half Integer	
0 1 1 0	Signed Vector 4x Quarter Integer	
0 1 1 1	Unsigned Vector 4x Quarter Integer	
1 0 0 0	Signed Vector 2x Quarter Integer	2x Vector Half Integer
1 0 0 1	Unsigned Vector 2x Quarter Integer	2x Vector Half Integer

C.8 Multiplexor

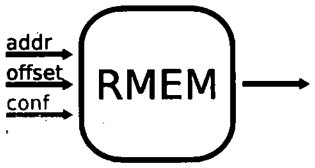


operation type	out = sel ? in1 : in2
conf bitwidth	4
SIMD	yes
arity	ternary

configuration

0 0 0 0	Single Integer	Equal Zero
0 0 0 1	Single Integer	Not Equal Zero
0 0 1 0	Single Integer	Greater Equal Zero
0 0 1 1	Single Integer	Greater Than Zero
0 1 0 0	Vector 2x Half Integer	Equal Zero
0 1 0 1	Vector 2x Half Integer	Not Equal Zero
0 1 1 0	Vector 2x Half Integer	Greater Equal Zero
0 1 1 1	Vector 2x Half Integer	Greater Than Zero
1 0 0 0	Vector 4x Quarter Integer	Equal Zero
1 0 0 1	Vector 4x Quarter Integer	Not Equal Zero
1 0 1 0	Vector 4x Quarter Integer	Greater Equal Zero
1 0 1 1	Vector 4x Quarter Integer	Greater Than Zero

C.9 Read Memory

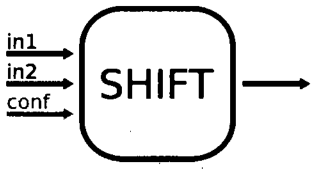


operation type	out = mem[addr+offset]
conf bitwidth	3
SIMD	yes
arity	binary

configuration

0 0 1	Single Integer	
0 1 0	Half Integer	Sign Extend
0 1 1	Half Integer	Zero Extend
1 0 0	Quarter Integer	Sign Extend
1 0 1	Quarter Integer	Zero Extend

C.10 Shift

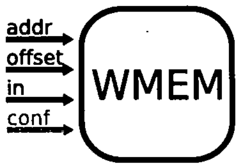


operation type	out = in1 $\ll$ / $\gg$ in2
conf bitwidth	5
SIMD	yes
arity	binary

configuration

0 0 0 0 0	Single Integer	Logic Left
0 0 0 0 1	Single Integer	Logic Right
0 0 0 1 1	Single Integer	Arithmetic Right
0 1 0 0 0	Vector 2x Half Integer	Logic Left
0 1 0 0 1	Vector 2x Half Integer	Logic Right
0 1 0 1 1	Vector 2x Half Integer	Arithmetic Right
1 0 0 0 0	Vector 4x Quarter Integer	Logic Left
1 0 0 0 1	Vector 4x Quarter Integer	Logic Right
1 0 0 1 1	Vector 4x Quarter Integer	Arithmetic Right
0 0 1 1 0	Single Integer	Left 31-1
1 1 0 0 0	Single Integer	Left 30-2
1 1 0 1 0	Single Integer	Left 28-4
1 1 1 0 0	Single Integer	Left 26-6
1 1 1 1 0	Single Integer	Left 24-8
0 0 1 1 1	Single Integer	Right 31-1
1 1 0 0 1	Single Integer	Right 30-2
1 1 0 1 1	Single Integer	Right 28-4
1 1 1 0 1	Single Integer	Right 26-6
1 1 1 1 1	Single Integer	Right 24-8

C.11 Write Memory



operation type	mem[addr+offest] = in
conf bitwidth	2
SIMD	yes
arity	ternary

configuration

0 0	Single Integer
0 1	Half Integer
1 0	Quarter Integer

Appendix D

Publications

Prime Author

Path Encoding	I. Nousias, S. Khawam, M. Milward, M. Muir, Y. Yi, and T. Arslan, "Encoding and Decoding methods," in *International Patent*, Pub. No. WO/2008/093123, International Application No. PCT/GB2008/000367, February, 2008.
Placement	I. Nousias, S. Khawam, M. Milward, M. Muir, and T. Arslan, "A multi-object ga based physical placement algorithm for heterogeneous dynamically reconfigurable arrays," in *Adaptive Hardware and Systems*, 2007, pp. 504–510.
RICA SMT	I. Nousias, S. Khawam, M. Milward, M. Muir, Y. Yi, and T. Arslan, "Simultaneous Multi-Threading on RICA," supplementary document, part of the *International RICA Patent*, Pub. No. WO/2006/114642, International Application No. PCT/GB2006/001556, April, 2006
VCDRC NoC	I. Nousias, S. Khawam, M. Milward, Y. Yi, and T. Arslan, "Wormhole routing with virtual channels using dynamic rate control for network-on-chip(NoC)," in *Adaptive Hardware and Systems*, 2006, pp. 420–423.

Main Contributor

RICA Journal	S. Khawam, I. Nousias, M. Milward, M. Muir, Y. Yi, and T. Arslan, "The Reconfigurable Instruction Cell Array," in *IEEE Trans. VLSI Syst. 16*, 2008, pp. 75–85.
Scheduling	Y. Yi, I. Nousias, M. Milward, S. Khawam, T. Arslan and I. Lindsay, "System-level scheduling on instruction cell based reconfigurable systems," in *DATE'06*, 2006, pp. 381–386.
RICA Patent	S. Khawam, I. Nousias, M. Milward, Y. Yi, and T. Arslan, "Reconfigurable Instruction Cell Array," *International Patent*, Pub. No. WO/2006/114642, International Application No. PCT/GB2006/001556, April, 2006

Contributor/Affiliated

Compression	N .Aslam, M .Milward, I .Nousias, T .Arslan and A. T. Erdogan, "Code Compression and Decompression for Instruction Cell Based Reconfigurable Systems," in *IPDPS*, 2007, pp. 1–7.
RTOS for RC	H .Wei, M .Muir, I .Nousias, T .Arslan and A. T. Erdogan, "Mapping Real Time Operating System on Reconfigurable Instruction Cell Based Architectures," in *FCCM'07*, 2007, pp. 301–304.
H264	A .Major, I .Nousias, S .Khawam, M .Milward, Y .Yi and T. Arslan, "H.264/AVC In-Loop De-Blocking Filter Targeting a Dynamically Reconfigurable Instruction Cell Based Architecture," in *Adaptive Hardware and Systems*, 2007, pp. 134–138.
Synthesis	Y. Yi, M. Milward, S. Khawam, I. Nousias and T. Arslan, "Automatic synthesis and scheduling of multirate DSP algorithms," in *ASP-DAC'05*, 2005, pp. 635–638.
AMBA	A. Olugbon, S. Khawam, T. Arslan, I. Nousias and I. Lindsay, "An AMBA AHB-based reconfigurable SOC architecture using multiplicity of dedicated flyby DMA blocks," in *ASP-DAC'05*, 2005, pp. 1256–1259.