Domain-specific and Reconfigurable Instruction Cells based Architectures for Low-Power SoC

Sami Khawam

A thesis submitted for the degree of Doctor of Philosophy
The University of Edinburgh
April 2006

Abstract

Silicon technologies have been conforming to the maxim of Moore's law for the past 40 years [131], but, even though production prices per unit have gone down, the NRE costs for making new chips keep going up with every new technology. This made a number of application-sectors discouraged to design new chips and in favour of adopting more generic solutions such as FPGAs and high-performance DSPs. These two programmable technologies have also evolved dramatically over the past decade providing much larger usable silicon areas and higher throughputs at the expense of increased power consumptions.

New communication standards and the requirements of modern mobile-device's users push the silicon towards processing more data in an increasingly shorter time; this is precisely the case for new compression formats targeting high-quality low-bandwidth multimedia. This presses forward the need for new programmable hardware solutions that intrinsically achieve generality, high-performance and, most importantly, low power consumption.

This work investigates the design of reconfigurable hardware architectures to address these issues. Two novel solutions are thus proposed along with the implementations of several multimedia applications on them; the first architecture fits as a middle ground between FPGAs and ASICs in terms of performance and cost. This is achieved by using coarse-grain functional units combined with programmable interconnects to build flexible, high-performance and low-power circuits. A framework for generating and programming the custom domain-specific reconfigurable arrays is also proposed. The tool-flow leverages some of the design effort that goes in creating and using the arrays by facilitating the reuse of previous design elements. Furthermore, this work proposes novel direction-aware routing elements to allow efficient tailoring of interconnect structures to the application.

The second proposed processing architecture adds the dimension of high-level programmability to the reconfigurable arrays. This is achieved by using functional units that can be directly matched to elements in a compiler's internal representation of software. By using a custom instruction-controller the array can execute control operations in a similar way to processors, while at the same time allowing highly efficient mapping of datapath circuits. Coupled to the low-power and high-throughput achieved, this creates a viable alternative to FPGAs, DSPs and ASICs suitable for deployment in high performance mobile applications entirely programmable using languages such as C/C++.

Acknowledgement

I would like to thank Prof Tughrul Arslan for his supervision during this project. I am most grateful for my parents George and Samia, whose support not only made this possible, but allowed me to actually enjoy my time throughout the process. I would also like to thank my sisters Lina and Zeina for their encouragement all way long. Special thanks go to my girlfriend Teresa for making this agreeable and for the tedious proof-reading; bad English should be blamed on her.

I also thank my examiners. Steve Furber, Ahmed Bouridane and Alan F Murray for their comments and advices.

Finally, I would like to thank everyone who gave me energy by expending their cooking: Teresa Kao, Sandy Gulyurtu, Khodor Fawaz and his mum, Ioannis Nousias, Nardine Osman, and Mehdi Tassoumt. I hope that this acknowledgement will only encourage every one of them to invite me more.

Table of Contents

List of Figures

List of Tables

Glossary / Acronyms

ASIC	Application Specific Integrated Circuit and commonly means the use of hardwired non-programmable silicon
AVC	Advanced Video Coding, otherwise known as H.264
Basic block	A block of instructions generated by a compiler where no instruction other than the first is jumped to, and no instruction other than the last one jumps to other locations
CLB	Configurable Logic Blocks are usually, in FPGA s, a group of several LUTs
CORDIC	COordinate Rotation DIgital Computer is an algorithm useful for the efficient calculation of trigonometric functions using a look-up-table, adds and shifts
DCT	Discrete Cosine Transform, time-to-frequency transform useful in image coding
Distributed Arithmetic	Calculation of a matrix-by-vector multiplication using look-up-tables, adds and shifts
DSP	A Digital Signal Processor is a processor with instructions useful for signal processing applications
DSRA	Domain Specific Reconfigurable-Array, the first fabric proposed in this work
DVB-T	Terrestrial part of the Digital Video Broadcasting standard for transmitting TV channels
eFPGA	Embedded FPGA, which is a programmable FPGA core than can be used as part of an SoC
FFT	Fast Fourier Transform, time-to-frequency transform useful in radio application
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
H.264	Video coding standard also referred to as MPEG-4 Part 10 Advanced Video Coding (AVC)
HDL	Hardware Description Language, a low-level programming language that describes parallelism. Examples: Verilog and VHDL
HLL	High Level Language usually for programming a processor, e.g. C, C++, Java
IC	Instruction Cell, the basic function units in RICA
Leaf functions	A function in a program that does not call any other function
LUT	Look-Up-Table, usually addressable memory with pre-computed data stored in it. In FPGAs, LUTs with programmable memory are used to create programmable gates
MCU	Micro Controller Units, can be seen as simple GPP with low mathematical processing resources and typically slower operating frequency
Motion	Calculation to find the temporal redundancy between two blocks in two consecutive

Estimation	video frames
NRE	Non-Recurring Engineering is the initial design effort and costs spent to allow the creation of end-units, irrespective of thee total number of units produced
OFDM	Orthogonal Frequency-Division Multiplexing, a radio modulation technique used in modern wireless standards such as DVB-T and WiMax
PLA	Programmable Logic Arrays
RICA	Reconfigurable Instruction-Cells Array, the second fabric proposed in this work
SDR	Software Defined Radio, a radio modem where the physical layer executes on a programmable fabric as opposed to the traditional way of using hardwired silicon
SIMD	Single-Instruction Mutliple-Data, a method used in a processor to increase speed in computations that are data parallel where the same operation is executed on a stream of data independently
SoC	System-on-Chip, an integrated circuit containing several cores which can be a combination of hardwired and programmable elements
VLIW	Very Large Instruction Word: a DSP processor with several operational units (typically 8) that are able to simultaneously execute independent instructions while sharing registers and memory.

Chapter 1:

Introduction

Undoubtedly, the traditional problems of hardwired Application Specific Integrated Circuits (ASICs) designs such as inflexibility and very high NRE costs – which have been increasing as the technology got smaller – have opened a big opportunity for reconfigurable technology to flourish. The typical use of software solutions such as processors and Digital Signals Processors (DSPs) for adding flexibility to ASIC designs is nearing its limits as new performance-demanding applications emerge. This is particularly true for new complex algorithms such as MPEG-4 and Advanced Video Coding (AVC) that require a throughput only achievable with high DSP operating-frequencies and high power consumption. Other solutions such as Field Programmable Gate Arrays (FPGAs) are able to achieve performance unattainable with conventional programmable systems such as (Micro Controller Unit) MCU and DSP processors, while providing an enormous margin of reconfigurability compared to ASICs. However, this flexibility comes at the cost of very high consumption power and silicon area, which makes them unusable in battery-operated devices. Figure 3-1 shows the characteristics of these discussed solutions. A current SoC implementation would ideally include several combinations of these solutions to meet requirements.

To solve this problem a multitude of research projects and commercial solutions have been proposed in several directions. One way to deal with these new requirements is to improve the performance of current processors and DSPs. This can be achieved by increasing the level of pipelining in the instruction issue and execution process, which boosts throughput for instructions with a sequential and predictable execution flow. However, this comes at the cost of wasting cycles when executing code contains conditional and unpredictable branch instructions. Another strategy for increasing performance is to execute several instructions in parallel as in Very Large Instruction Word (VLIW) and Superscaler processors. This usually gives a good performance enhancement when compared to single-issue processors; however, in VLIWs only independent instructions can be executed simultaneously and the problem is that typical programs are not abundant in instruction level parallelism (ILP), which creates a practical barrier to the extent of achievable performance. Although all these DSP-based solutions offer very good flexibility, they usually have much less performance and a lot more power consumption than hardwired ASIC solutions. The current ongoing trend for increasing performance in processors is to have multiple cores that are able to execute multiple threads simultaneously. Although this is a very promising approach, it still requires a lot of effort to radically change the way programs are written and compiled so that parallelism is explicitly or implicitly defined.

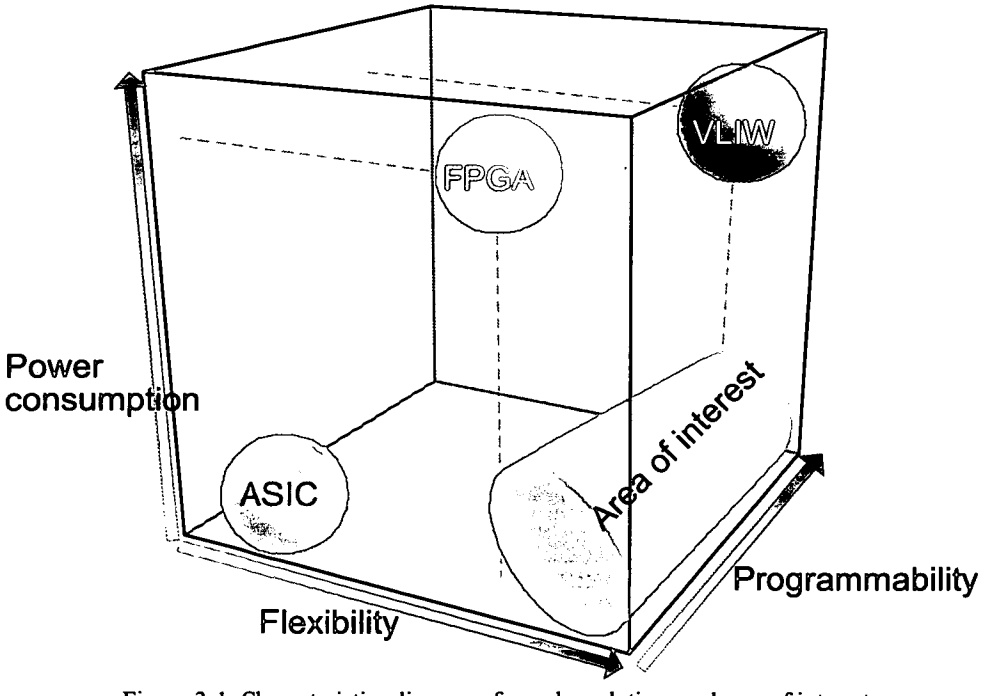


Figure 3-1: Characteristics diagram of popular solutions and area of interest

The popular reconfigurable logic and Field Programmable Gate Arrays (FPGAs) today offer very high flexibility compared to ASICs and higher performance than DSPs - hence they

represent a potential architecture for future implementations. In a similar way to ASICs, the high performance of FPGAs comes from the fact that they have the ability to implement a large number of parallel operations on their fabric. The main drawback in FPGAs is their very high silicon area and power consumption which makes them unusable in portable and battery-operated devices. Also – similar to ASICs – FPGAs are programmable using a Hardware Description Language (HDL) as opposed to processors that use high-level languages such as C/C++. In HDLs the parallelism between operations is explicitly defined, where as languages such as C have traditionally been used for serial definitions of operations. Nevertheless programmability through high-level languages is preferred over HDLs since high-level languages are more popular as many existing designs and new standards use them. Furthermore, programming at HDL-level requires much more effort for representing algorithms in a parallel form. An easy *programmability* is crucial for the success of any hardware architectures as it reduces the design-time and time-to-market.

As opposed to the single-chip FPGA solutions, embedded FPGAs (eFPGAs) are reconfigurable logic cores that can be fitted inside a custom System-on-Chip (SoC) to increase its post-fabrication flexibility. Several commercial eFPGAs exists, even though they still suffer from high area and power overheads. Their usage is also problematic as it complicates the overall chip design tool-flow at the verification and implementation stages.

While FPGAs are mainly a lump of programmable gates, there is currently a trend of so called *reconfigurable computers/architectures* which recently gained two types of definitions. The exact detail of the inside of a reconfigurable computer can be some combination of a processor and FPGA fabrics, such as the case where an array implements the processor's ALU to effectively allow reconfiguring the processor's instruction-set. Reconfigurable computers can also be seen as a fabric with special programmable elements for which software can be compiled in a similar way to processors. There are several proposed architectures that fall in this category promising high performance gains by using FPGA-like parallelism, while at the same time providing the ease of use found in processors. Figure 3-1 and Figure 3-2 show the different advantages and disadvantages of the existing SoC solutions; reconfigurable computers are promising to fill the performance and flexibility triangular gap between DSP, FPGAs and ASIC. As detailed in Chapter 2, most of the existing architectures suffer from disadvantages in flexibility, performance, programmability or area and power overheads. It can also be noted that most architectures were designed to have the highest performance possible while maintaining good flexibility and hence there is no solution that tackles the power consumption problem specifically.

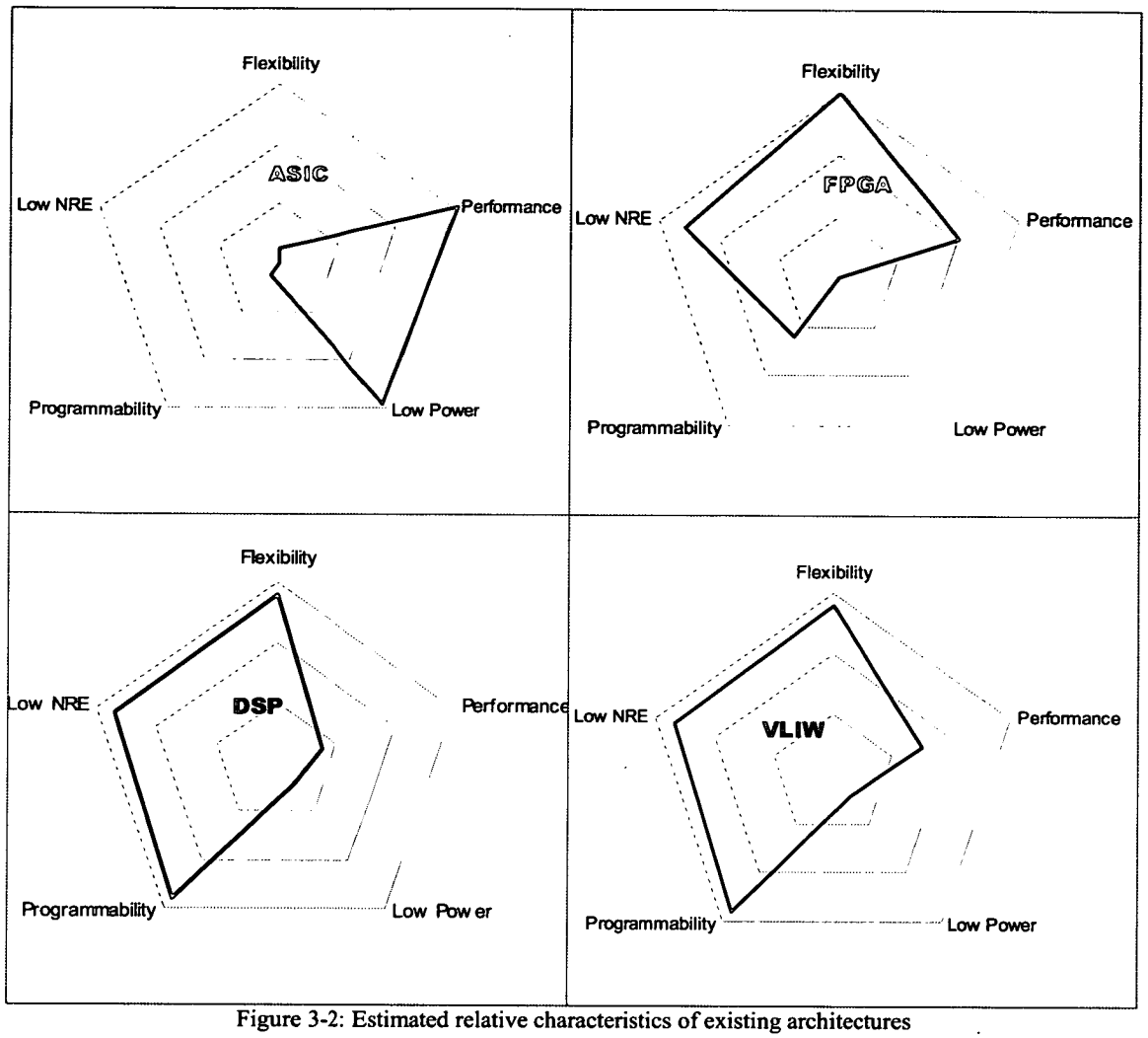


Figure 3-2: Estimated relative characteristics of existing architectures

This thesis explores these reconfigurable technologies and tries to extend the existing architectures to find a solution for future portable devices. Here, we are trying to prove that it is possible to efficiently exploit the "area-of-interest" highlighted in Figure 3-1 in order to find an architecture that gives a better throughput than current programmable technologies, while achieving much lower power consumption and/or better programmability. This is explored here using two approaches: domain-specific arrays and instruction-cell arrays. The comparison of the performance of these approaches with existing and ideal solution is shown in Figure 3-3 and Figure 3-4.

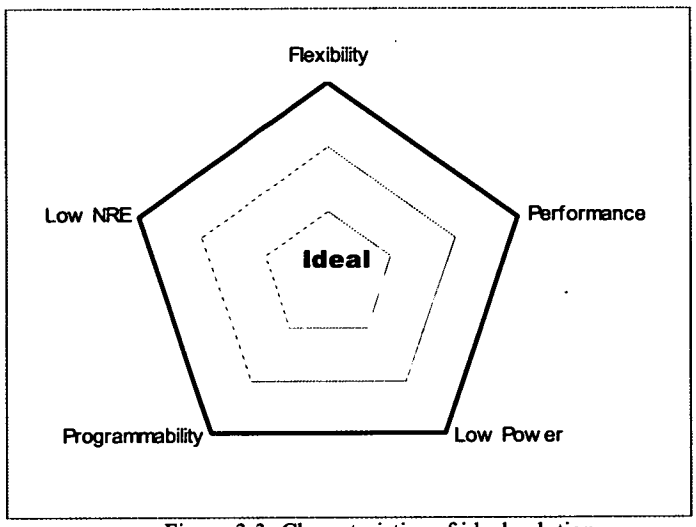


Figure 3-3: Characteristics of ideal solution

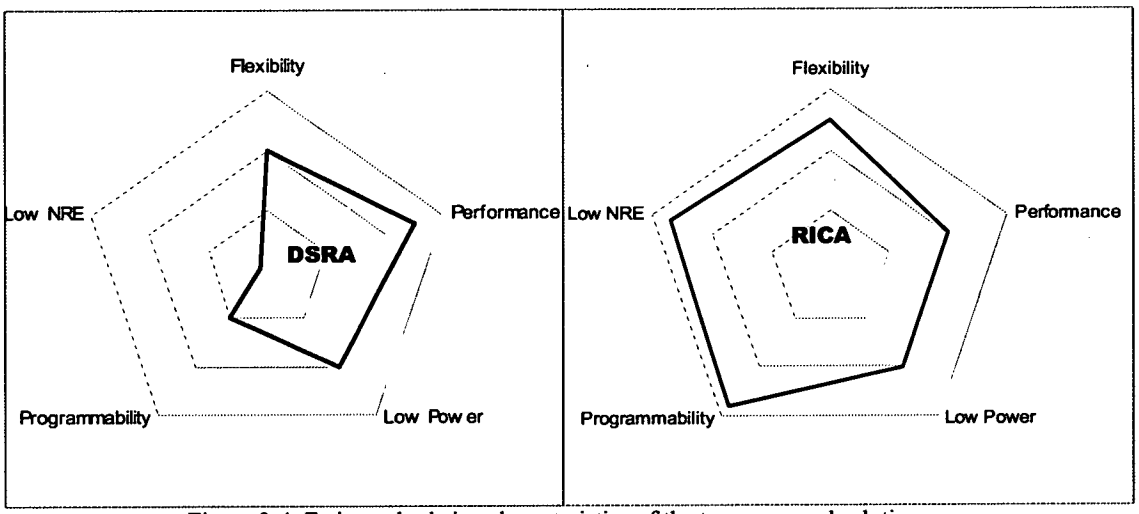


Figure 3-4: Estimated relative characteristics of the two proposed solutions.

The domain-specific arrays (*DSRA*) are based on the observation that in most SoCs the design that would be mapped to an eFPGA is chosen at the partitioning stage prior to the design of the hardware and that, depending on the application, only a specific portion of the eFPGA is usually used completely for random logic. This opens the opportunity to use an eFPGA that is more domain-specific to the target application but which has increased performance in power, timing and area when compared to generic eFPGA. This is usually achieved by using coarse-grain programmable elements as opposed to the fine-grain ones in FPGAs. Although such a domain-specific solution can be extensively designed for every application encountered, a rapid generation of such architectures is essential to have a usable programmability. Hence, the initial approach described in Chapters 3, 4 and 5 proposes the so called Domain-Specific Reconfigurable Arrays (DSRAs) to semi-automatically create SoC cores that achieve good performance, area and power consumptions while at the same time providing a margin of flexibility to support post-fabrication changes, as seen in Figure 3-4. The DSRA approach

proves that it is indeed a compromise between ASICs and FPGAs as it can achieve up to 3 times less power and 60% less area than an FPGA, while having 3 and 2.5 times more area and power than ASICs. A methodology for creating and using such cores inside an SoC is proposed, along with optimised implementations of multimedia operations.

However, the DSRA approach inherits the low programmability found in FPGAs, since it tries to port ASICs and FPGA designs to the architecture while reducing power and area. Chapter 6 introduces the Reconfigurable Instruction Cell Array (RICA) where the design of the hardware fabric is in such a way that it can accept a high-level description of a program. The RICA can be viewed as a coarse-grain array that can be programmed in a similar way to processors. Due to its array structure and abundant processing elements, RICA provides more parallel processing than high-end DSPs, while at the same time it consumes lower energy. Results show that RICA can be around 10 times faster than VLIW DSPs at a 6 times lower power consumption in the datapath. Furthermore, as described in Chapter 7, big systems such as full H.264 video-decoders can be quickly and easily mapped to RICA simply by using an existing C program description.

Chapter 2:

Previous Reconfigurable and low-power architectures

With the high costs of current and future chip design and manufacturing technologies there is an urgent economical need to reduce the number of required re-spins in a design and to extend the life of manufactured devices. This can generally be achieved by adding flexibility and programmability to Application Specific Integrated Circuits (ASICs), which allows making changes to the design after manufacturing in order to overcome design errors and/or to support new and updated standards. The flexibility also allows dynamic reconfiguration which helps the system adapt to run-time constraints to improve the performance. Such flexibility is currently achieved using software solutions; however, the use of processors and DSPs in performance-critical applications such as portable devices is not beneficial. This is particularly true for new complex algorithms such as MPEG-4 and Advanced Video Coding (AVC) that require a high throughput only achievable with a high DSP operating frequency and high power consumption.

On-going work to find better architectures for future devices has led to several novel systems upon which the work presented in this thesis is based. Existing and established architectures described in the previous chapter like DSPs, FPGAs and ASICs are listed in Table 3-1. The rest of this chapter will detail the features of all emerging and researched reconfigurable technologies. As will be shown later, only a few of these architectures can potentially provide suitable high performance and low-power consumption. The pros and cons of every architecture are described to allow drawing a comparison between the solutions.

Table 3-1: Established solutions

ASIC	
Pros:	High speed, Low power
Cons:	Low flexibility, high NRE costs, designed using HDL
FPGA	
Fabric:	Fine-grain look-up-tables (LUT)
Interconnects	Symmetrical Mesh
Pros:	Very high flexibility,
Cons	Very high power consumption, programmable using HDL
DSP, low-power DSP, VLIW, Superscaler, SIMD	
Architecture:	ALU-based. Can take advantage of Instruction Level Parallelism
Pros:	Programmability using high-level languages, high flexibility
Cons:	Limited throughput
Multi-Core and Multi-processor	
Architecture:	Multiple cores with multi-threading between core to increase parallelism
Pros:	High throughput, programmability using HLL
Cons:	Synchronisation between the cores currently requires manual work.

This chapter first examines reconfigurable logic structures and reconfigurable computing architectures, i.e. systems able to execute a program-like sequence of instructions. Since programmable interconnects represent a big contribution to flexibility of reconfigurable systems, and consequently a considerable part of this work focused on the interconnects, the second section of this chapter overviews the existing programmable interconnects topologies.

2.1. Reconfigurable arrays and computers

Reconfigurable arrays can be generally defined as programmable fabrics where a circuit/datapath is mapped for execution. Even though the arrays might support partial dynamic reconfiguration, we define *a reconfigurable array* any situation where the datapath mapped is fixed temporally; the circuit usually contains its own control and datapath elements. Reconfigurable arrays can be further classified into ones based on fine-grain or coarse-gain elements as functional units.

Another class of reconfigurable architectures includes structures programmable to execute both control and datapath operations. This can be further split into reconfigurable processors which are simply a tight combination of an FPGA and a processor and reconfigurable

computing architectures, which are fabrics that can directly execute control and datapath operations.

2.1.1. Fine-Grain arrays

Commercial FPGA architectures, such as [1] and [2], are fine-grain arrays, as this gives the maximum flexibility possible. The operational elements are the Configurable Logic Blocks (CLBs) which are mainly Lookup-Up-Tables (LUTs) with 16 single bit inputs. These inputs are controlled by the bits from the configuration memory, making it possible to build any 4-input logic function by changing the content of the SRAM configuration memory [41]. The programmable elements also have the ability to register their outputs. Furthermore, a mesh of programmable interconnects is available to connect the CLBs together to build bigger circuits. The structure of these single-bit level interconnect is described below in Section 2.2. The fine-grain aspect of FPGAs makes them extremely flexible and suitable for a very wide range of application. Hence, FPGA chips are produced in large quantities which makes their usage come with very reduced NRE costs. This high flexibility also implies very high power consumption which prohibits the deployment of FPGAs in portable applications. In terms of performance FPGAs have usually around 10 times longer delays than ASICs. In an FPGA chip the energy dissipated in interconnects is about 65% of the total energy consumption, while 30% are dissipated in programmable clock-routings and IO blocks [4].

Although FPGAs are traditionally homogenous arrays of fine-grain CLBs, some FPGA manufacturers recently started adding large application-specific blocks inside the fabric, such as multipliers, arithmetic operators and general purpose processors [1].

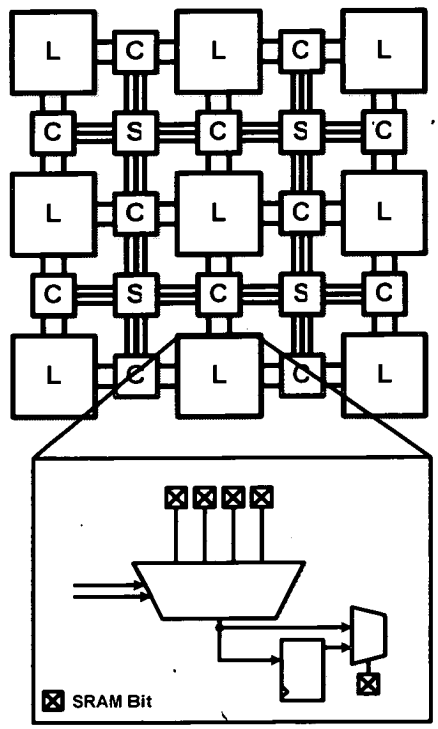


Figure 3-5: Example topology of an FPGA showing a simplified 4-to-1 LUT.

In order to add flexibility to custom ASIC and SoC designs, FPGA technology can also be used as embedded FPGA (or *eFPGA*) cores. As in single-chip FPGAs, eFPGA cores contain the same array of programmable LUTs and an interconnect network. Existing commercial eFPGAs are described in [5]. They represent a good development towards programmable custom SoCs, however, designers are faced with problems due to the difficulty of integrating these analogue-level cores into SoC. The existence of a big programmable hard-core in the SoC makes tasks such as verifications, timings and power analysis difficult, as the characteristics of the core are very dependent on the design mapped on it. Furthermore, the existence of such configurable transistor-level IPs in the SoC makes the overall implementation tool-flow complex.

To overcome this problem, embedded synthesisable reconfigurable logic was proposed in [6] where synthesisable programmable logic to implement combinatorial functions such as next-state circuits based on programmable Look-up-tables (LUTs). The elements are spread in the circuit and are suitable for small logic functions and glue-logic between the bigger elements of the SoC. The area of the circuit in [6] is larger than the area of normal FPGAs due to the use of synthesisable cells.

Table 3-2: Improvements to FPGAs

Synthesisable FPGA [6]	
Fabric:	Based on LUTs to build small logic functions and glue-logic.
Performance:	The area is larger than FPGA due to the use of synthesisable switching circuit elements.
FPGA with Dynamic Reconfiguration: DP-FPGA [7]	
Fabric:	Similar to a fine-grain FPGA, but supports fast dynamic reconfiguration by storing multiple context in the FPGA memory.
Performance:	The ability to support fast dynamic reconfiguration was found to increase the silicon utilisation of an FPGA by 3-4x times.
Low Power FPGA [4]	
Fabric:	Fine-grain LUT based fabric, but with modified interconnects and clock routing circuits to reduce the power. Very low-level and non-synthesisable techniques are employed.
Performance:	This architecture presents an order of magnitude improvement, in terms of power, over commercial FPGAs, while still maintaining the same speed.

Another problem with FPGAs is the large number of configuration bits they require (typically in the order of 5 MBits for recent devices [1]), which makes the time required to program these bits long. This can be a restriction if dynamic reconfiguration is desired in cases where parts of the circuit mapped on the FPGA are idle waiting for another part to finish. Dynamic reconfiguration of the circuit in this case would lead to better use of the available silicon. To enable this, FPGA manufacturers started allowing partial reconfiguration of the device, which would take a relatively short time to reprogram as long as the area reconfigured is small. On the other hand, the DP-FPGA project [7] proposed an FPGA architecture that can store

multiple configurations and switch between them. Even though the memory area needed to store the configuration is large, this approach was found to increase the silicon utilisation of an FPGA by around 3-4 times.

An attempt to reduce the power consumption of FPGAs was proposed in [4] and included a combination of analogue circuit techniques and interconnect topologies. The approach in [3] and [4] was to reduce the power dissipated in interconnects and in the clock-trees. Even though the power dissipated in the CLBs is negligible, their structures were slightly modified to provide a better overall routing capability to suit the interconnect topology (described later in Section 2.2). On the circuit level, low-swing circuits are placed on both ends of an interconnect line to reduce the voltage swing to 0.8V, while the rest of the circuit runs at 1.5V. This reduction in voltage improves power consumption. The power dissipation in the global clock distribution networks is reduced by using dual-edge triggered flip-flops in the CLB, which halves the operating frequency, however, it puts more constraints on the clock signal generator (e.g. correct duty-cycle). A 0.8V voltage swing is also used in the clock trees. This architecture presents an order of magnitude improvement, in terms of power, over commercial FPGAs, while still maintaining the same speed. The area is only increased a small amount due to the added circuits. However; the above-mentioned circuit level techniques would be difficultly to implement in an embedded FPGA and hard to integrate into an SoC. Such circuit level techniques become very complex especially when trying to create a synthesisable core, as that means that new library cells have to be manually created.

2.1.2. Coarse-Grain / Domain-Specific arrays

The efficiency of implementing an algorithm on FPGA hardware greatly depends on the structure of the basic logic-block used in the array. As described above, commercial FPGA implementations provide a fine-grain structure that can be used to implement a wide range of hardware. However, this generality adds hardware overheads such as interconnects, which affect the power, speed and area efficiency of the implementation. By making hardware architectures less generic and more specific to a domain of applications, several improvements can be gained in terms of power efficiency, speed and area.

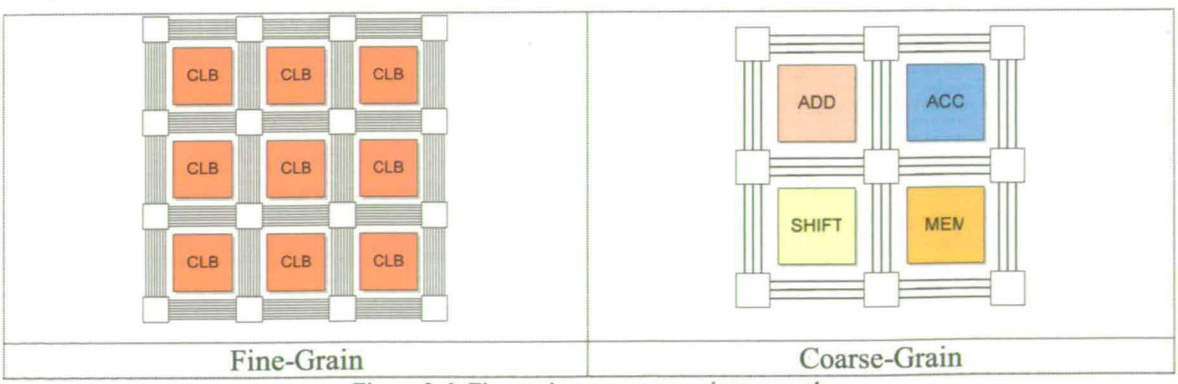


Figure 3-6: Fine grain vs. coarse grain approach

As shown in Table 3-3 below, several commercial and academic coarse-grain arrays exist; the CHESS architecture from [8] is an array of 4-bit ALUs targeting general multimedia applications. The array proposed in [9] is based on 4-bit LUTs with reduced flexibility in implementing random logic leading to a smaller area. The commercial D-fabrix from Elixent [10] is another attempt to reduce the area and power overhead. Although this approach is efficient, it still requires low-level manual coding for mapping the implementations.

Table 3-3: Coarse-grain arrays

D-fabrix/Elixent [10] (Similar: [8] and [9])	
Fabric:	Homogeneous grid of 4-bit ALU units. This ALU bit-width is not high enough to be defined as coarse-grain, but it is wider than the 1-bit in FPGAs. The array works as a coprocessor and the synchronisation between the host and the array has to be done manually.
Programmability	Programming the array is done at hardware netlist level using Handel-C or VHDL.
Array customisation	The array is not synthesisable and hence difficult to port to new process technologies.
Performance benefits:	Timing and power comparison to other solutions are not disclosed.

Another example of efficient domain-specific PLAs has been shown [11]: An FPGA architecture is proposed for the implementation of reduced complexity filters using a Primitive Operator Filter (POF). POF uses primitive operators such as *shifts*, *additions* and *subtractions* in the form of signal flow graphs to replace multiplications in digital filters. Thus, different CLB structures are described and compared. The CLBs consist of shifters, adders and subtracters to implement POF structures, as well as latches for memory elements and multiplexers. The multiplexers are used to route signals inside the CLB and to select the output signal of a CLB. Different CLB granularities are investigated and their performance compared in terms of speed and area. Since the CLBs are all connected to a single data bus, the speed of the output throughput is limited. In [12], a similar PLA architecture is presented, but with local reconfigurable interconnects between the CLBs, similar to the ones in commercial FPGAs. However, the advantage of using this structure over generic commercial ones is that the overall number of interconnects is much lower and, thus, the area and delays are reduced. This structure is also more power efficient since less power is dissipated in the interconnect.

2.1.3. High-level FPGA synthesis

Several attempts have been made to increase the programmability of FPGAs, trying to automatically synthesise programs written using high-level languages into FPGA circuitry. The first class of such tools use programming languages having a higher description level than HDLs; this is the case of the SA-C language provided by the Cameron project [13] and

Handel-C provided by Celoxica [14]. Although these languages are easier to use than standard Verilog and VHDL, they still represent only a small subset of the standard ANSI-C and they have their own non-standard constructs, which prohibits reusing code written in standard C.

Table 3-4: High-level synthesis of FPGA circuitry

FPGA with SA-C [13], Handel-C [14]	
Programmability:	SA-C is a subset of ANSI-C without pointer and where variables represent wires. In the Cameron project which uses SA-C, VHDL is still required to make the control logic. Handel-C is also a subset of C and requires existing C program to be re-written to explicitly define parallelism between functions.
Performance:	Using these languages typically leads to 20% performance degradation over the manual design of the FPGA circuit in HDL.
FPGA with FREEDOM [15] and [16]	
Programmability:	Compiled binaries (which can be generated from any high-level language) are converted into a number of FSMs that are mapped to the FPGA.
Performance:	A speedup of 1.3-5x was observed between the FPGA exeution (on Xilinx Virtex 2) and the DSP execution (an TI C64x VLIW).

The FREEDOM compiler from Binachip [15] [16] is a more successful attempt to create FPGA circuitry from existing program binaries, which can be created by compiling a high-level program source. The program binary, which represents a Control Flow Graph (CFG) of scheduled instructions, is converted into a number of Finite State Machines (FSMs) that are executed in sequence on the FPGA to achieve the same operation.

2.1.4. Reconfigurable instructions-set processors

Reconfigurable instruction-set processors can tailor the possible operations executed each cycle by the processors elements (e.g. ALU) according to the application. This can for example be the creation of an ADD-SHIFT instruction which combines 2 ALU operations in a single cycle, if the application uses this pair of operations frequently.

Table 3-5: Reconfigurable instruction-set processors

Configurable instructions (Chimaera [17], ConCise [18], Tensilica [19])	
Fabric:	Processors with reconfigurable fabric embedded into their pipeline which allows creating customised instructions.
Programmability	Full ANSI-C, the compiler only has to know about the extra instructions added.
Performance	The problem in such processors is that they cannot achieve a very high throughput, as they are still limited by the typical problems of processors.

2.1.5. Loosely and tightly coupled arrays and processors

Reconfigurable processors are a combination of a processor and a reconfigurable FPGA-like structure, where all the compute intensive operations are executed on the FPGA to gain

improvements. A large number of such processors exists [20]. Such architectures suffer from the fact that a lot of manual work goes into designing the code for the processor and the reconfigurable fabric – which in most cases has to be done separately. Furthermore, data and time synchronisation between array and the processor requires manual interference.

Two classes of such systems can be distinguished according to the loose or tight coupling of the array with the processor.

Table 3-6: Loosely coupled processor and a reconfigurable array

Garp [21]	
Architecture:	A fine-grain array with 2-bit CLBs acting as a coprocessor to a DSP. The array and the processor communicate using a shared memory block. The processor is responsible for configuring the array and for synchronising the operations time with the array. The configuration time is relatively slow as it requires the transfer of 6 kbytes, however, this is still faster than the time needed to configure an FPGA.
Programmability	The program for the array is created using a proprietary netlist language, independently of the program running on the processor, which takes care of the synchronisation.
Performance	Depending on the application, speedups between 2 and 24 times were observed when using this coprocessor, which is quite typical of speedups obtained between FPGAs and processors.
Low-power	Not disclosed
Morphosys [22]	
Fabric:	A RISC Processor coupled to a homogenous coarse-grain array of 32-bit ALUs (containing a multiplier and a register file). This architecture follows the SIMD model, since all the functional units in the same row or column execute the same operation but on different data. Hence the array is only useful for data-parallel operations, while the rest of the (control) operations are executed by the RISC. Its main target is pixel-processing where such parallel-data operations are common. Data transfer to/from the array is programmed manually into the RISC, along with all the required synchronisation between the two. One advantage is that the array and the RISC can both be functioning at the same time.
Performance:	In operations such as DCT, Motion Estimation and Viterbi-decoding around a 5-10 improvements over normal CPUs is observed.
Programmability	Both the RISC and the array are programmed using low-level assembly language.
Customisation	Although the core is synthesisable it is not customisable.
Low-power	Lower power over DSPs is claimed, details not disclosed.
Recore Systems's Chameleon/Montium [23]and [24]	
Fabric:	The coarse-grain array acts as a co-processor to a general purpose processor in order to execute datapath code (no control). Several arrays (the proposed example has 4) can be used together through an interconnect scheme. The processor is responsible for configuring and operating all the arrays. It has the potential to achieve high bandwidth through parallel and distributed memory access.
Programmability	Proprietary Montium LLL language which is quite low-level.
Low-power	Benchmarks with other solutions are not disclosed.

SiliconHive [25]	
Fabric:	Arrays of Processing and Storage Elements (PSE) cells built around a base processor. The base processor handles control applications and distributes datapath operations to the PSEs. Example PSEs from Avispa-CH1 (for SDR application) are DSP units supporting complex arithmetic.
Interconnects	Done between cells using blocking FIFOs accessed from each cell.
Programmability	All the processors (base processor and PSEs) are programmed using standard C language, however, the timing and data synchronisation between them has to be coded manually..
Array customisation	The architecture is synthesisable, scalable and different types of PSEs can be used.
Low-power & performance	Not disclosed
PACT from XPP Technologies [26]	
Fabric:	The XPP64-A1 chip is built from an 8 x 8 array of ALU-PAEs (Processing Array Elements) with 2 rows of RAM-PAEs at the edges (each has 512 x 24 bit). The core supports general-purpose opcodes and special operation such as packed complex arithmetic. Programs are partitioned into datapaths for the PAE and control operations for the host processor
Programmability	Special NML language, which is quite low-level and difficult to program.
Low-power	Not disclosed
REMARC [27]	
Fabric:	Coarse gain 8x8 array of 16-bit *nanoprocessors*. Coupling between RISC and fabric is done through registers, with some registers shared between both (which can be defined as tight coupling).
Performance:	This approach was compared to the use of a processor with an FPGA array, and it was found that a coarse-grain REMARC array of the same size gives around 7 time better performance.

Table 3-7: Tightly coupled processor and a reconfigurable array

Matrix [28]	
Fabric:	Similar to MorphoSys as being a combination of having a RISC and an array, but in this case they share the same configuration memory. Quite old, has no multipliers and targets simple operations. Functional units are 8-bit ALUs with memory and some control logic.
Performance and Low-power	Not disclosed
PipeRench [29]	
Fabric:	The array consists of a series of stripes each containing programmable ALUs that are interconnected using programmable pipeline stages in order to implement highly-pipelined datapath circuits. A feature if this architecture is the ability to reconfigure every block in one clock (the configuration is stored in context memory). Thus, e.g., a computation that requires 5 different operations in series can be implemented using only 3 blocks by constantly changing the configuration at each cycle in a pipelined manner (stages are configured while others are executed).
Interconnects:	Data connections are only present between two consecutive ALUs, in such a way that the output of the previous block is fed to the input of the next one. The processor and array communication is done through a FIFO.
Programmability	Uses a special language which is a subset of C that only supports single assignments. When compiled programs are converted into a straight-line single-assignments by inlining all the functions and loops – hence the applications are limited to non-control ones.
Low-power	Not disclosed
ADRES [30]	
Fabric:	A VLIW coupled with a coarse-grain array. Memory and registers are shared between the array and VLIW to simplify the programming model of this processor/co-processor scheme – the only difference is that the register file is shared. A datapath on the array can support limited control operations: if a loop requires small conditional executions they get converted into *predication* (i.e. conditional execution). The configuration RAM stores several contexts to allow fast switching between them – this is also extended by the ability to load extra configurations from the system's main memory.
Programmability	Through C, since array and VLIW share memory and registers. Loops which can be pipelined and fit onto the array are automatically identified and mapped to the array. Data communication between the array and the VLIW is automatically done through the registers.
Performance:	Around 3x faster than a VLIW when mapping an application such as a MPEG-2 video decoder.
Low-power	Not disclosed

2.1.6. Reconfigurable computing architecture

Although some of the architectures described below in Table 3-8 can be seen as yet another combination scheme of a processor/microcontroller with an array of Functional Units (FU), *reconfigurable computing architectures* in general are more a solution where both control and datapath computations are naturally executed on the same fabric without the need for moving a large amount of data or manually synchronising the operation of the different elements.

Table 3-8: Reconfigurable computing architectures

RAW [31]	
Fabric:	Array of 16 *tiles*, where each tile is a processor coupled with some FPGA-like reconfigurable circuitry. Current RAW architecture targets high-end processing architectures as each processor has a Floating Point Unit. Each processor has its own instruction memory (and cache) and can access several banks of data memory.
Interconnects	Big programmable network of switches to connect each tile to its neighbours.
Programmability	On going work on a C compiler that allows high-level programming taking advantage of several levels of parallelism such as Instruction and Thread Level Parallelisms. However, current optimised implementations require manual low-level coding.
Performance	Hand-written and parallelised code achieved a performance comparable to FPGAs [32].
Low-power	RAW targets high-end processing and power reduction measures are not implemented. The area is a massive 255mm^2 on 0.15μm.
Pleiades [35] [36]	
Fabric:	Coarse grain satellites (e.g. 16-bits) units around a main processor. The main processor executes control-dominated sections of the program while satellites execute data-dominated computations. The system is distributed in a sense that every satellite has its own instruction fetch and execute. The satellites communicate between each other through dedicated interconnects. The satellite processors could be arithmetic modules (multipliers, MACs, etc.), memory modules, address generators or reconfigurable arrays.
Programmability	The design of the architecture and the choice of satellites to use have to be done manually. At partitioning stages the designer decides which loops of the full high-level program need to speeded-up using reconfigurable fabric; then the choice of deployed satellites can be made and their design started. This technique can create efficient architectures, however, they become too specific to the application. Programming the satellites requires writing low-level netlists.
Interconnects	See Section 36.0.5 below.
Array customisation	Interconnects and the type/number of satellites can be made tailored for the application.
Low-power	Not disclosed

Totem/RaPiD [33]	
Fabric:	RaPiD is a linear 1D array of coarse-grain Functional Units (FU). FU are of the order of ALUs, multipliers and shifters. It can implement dataflow graphs where the result of one FU is forwarded directly to other FUs that use it. The intermediate values are stored in distributed registers. The hardware allows two levels of configuration switching: A fast one that can change every cycle and a slower one (the decision is made at programming time by the compiler). A sequencer acts as a program controller to the array for loading and decoding the configuration – a standard RISC ALU is also provided inside the sequencer to execute control-like instructions that are not suitable for the FUs.
Interconnects	Pipelined data buses between the functional elements. Data buses restricts the scalability, as the number of FUs can only be increased if data locality is maintained, which requires a lot of design efforts.
Programmability	Uses RaPiD-C which, despite the name, is an assembly-level language that allows describing multiple parallel threads. All the synchronisation between threads is manually programmed using signals. However, the compiler automatically performs the pipelining and retiming required. Programming the RaPiD requires a detailed knowledge about the underlying reconfigurable fabric
Flexibility and array customisation	To achieve high throughputs for certain applications, a new array has to be generated with appropriate FUs, since each RaPiD array is not generic enough to support all applications with a high throughput. In the Totem project, research is also being carried out for the automatic generation of custom FUs, interconnects and VLSI layout of the core by specifying the high-level C algorithms [132].
Performance:	For OFDM [34] application, around 6 times speed improvement over VLIW DSPs was observed.
Low-Power	Not disclosed
TTA [38] [39]	
Fabric:	Uses general Function Units (FU) such as ALUs and register files combined with Special Function Units (SFU) that execute application-specific computations. Units are all pipelined in order to improve the performance of repetitive loops, which is the target application of this architecture – the TTA architecture is well suited for small applications such as DCT, Viterbi-decoding and encryption.
Interconnects	Based on a bus with segmented tracks. Although the design of the bus is simple, it limits the scalability of the system: The arrays have to be limited to small number of units (in the order of 25).
Programmability	Standard ANSI-C is supported. However, as with any processor, some manual assembly code is required to achieve high throughput and to make sure the timing in highly pipelined loops is met.
Array customisation	The arrays have to be customised to every application, since it is not possible to create a big array containing enough units to achieve high-throughput for every application.
Performance	Good ratio of area / throughput is achieved: High speeds can be achieved for the amount of silicon area used, however, in some applications an ARM9 processor can achieve a higher speed than TTA at the cost of higher area, which, in a way, limits the application of TTA in future devices.
Low-power	Not disclosed, only area consumption is measured.

2.1.7. Generic low-power solutions

Only a few of the previous research projects specifically target reducing power consumption, as the majority are concerned with achieving high performance. Furthermore, only a few of the previous works focus on developing generic signal-processing architectures with reduced power consumption, since it is easier to achieve power reduction by tailoring the hardware to the application. This includes low-power DSP processors such as the Hi-Perion from Fujitsu [40], which has the flexibility of normal DSPs but with lower power consumption. To achieve this it uses application-independent techniques, such as physical improvements in size and circuit capacitance as well as standard methods such as pipelining and parallel MAC processing to improve the performance and hence lower the supply voltage / operating frequency.

2.2. Interconnect structures in FPGAs

In an ideal situation where a reconfigurable system has Functional Units (FUs) operating in parallel, every FU would be able to connect to any other FU to exchange data. Although this is useful, it is quite often expensive in terms of area and power consumption. Since not all FUs need to be connected to each other at any one instance of time or in any single application, an interconnect scheme – depending on the FU type/structure/data handling – can be used to reduce the overall area and power usage. This section lists interconnect scheme used in FPGA devices, which have also been reused in other reconfigurable architectures.

2.2.1. Symmetrical Mesh

The symmetrical mesh architecture, which is also referred to as the island-style interconnect, is a popular structure found in most commercial FPGAs, which are characterised by a large number of homogenous logic-units that are commonly connected 'randomly' together. The logic blocks are grouped into clusters of blocks [41], generally containing between 4 to 10 modules (these clusters are sometimes called *slices*). Each cluster contains internally another layer of interconnects between the modules themselves. As shown in Figure 3-7, the array has fixed horizontal and vertical metal tracks run between the clusters and two types of configurable switches are present: *Connection-boxes* permit the connection of a pin from the cluster to the metal tracks, and on every crossing of the metal tracks a reconfigurable *Switch-box* connects the tracks together.

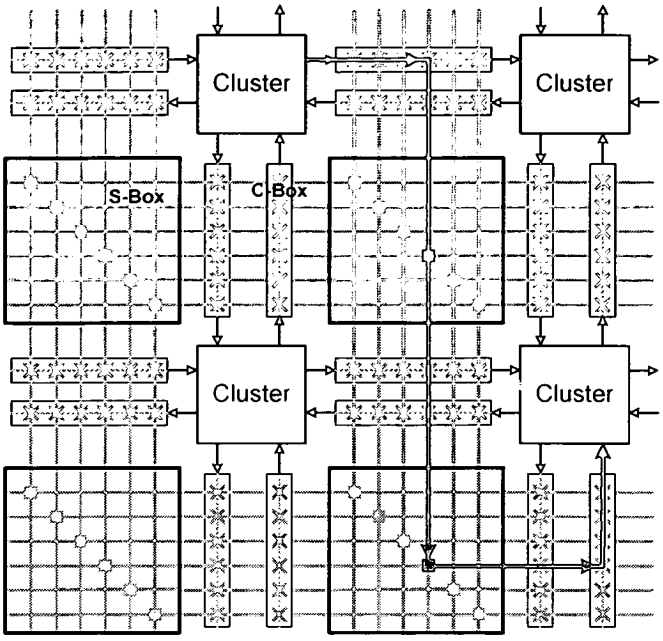


Figure 3-7: Signal routing between two clusters using switch and connection boxes.

The internal design of these reconfigurable switches and interconnect elements affects the overall flexibility and power consumption of the array. The *flexibility* of a switch or connection box is determined by the number of possible programmable connections as defined in [43] [42]. The flexibility of these boxes affects the overall flexibility of the array (hence routability) as well as other characteristics such as area and power consumption. As shown in [44], the design of the boxes is dependent on the type of logic blocks used.

In [43] Rose and Brown concluded from place and route experiments with multiple designs that FPGA connection blocks need high flexibility to achieve a high percentage of routing completion, and that relatively low flexibility is needed in the switch blocks. In commercial FPGAs the programmable switching circuits inside the boxes are implemented using pass-transistors, tri-state buffers or multiplexers.

Several topologies for the S-Box designs exist and their performance tends to be related to the type of the logic cells and the application mapped to the FPGA. The main topologies are the Disjoint [52] (used in Xilinx, also called subset), Universal [51] and the Wilton [53]. The work in [49] also proposes an s-box topology to support non-rectangular array forms. This would particularly be useful for embedded configurable logic, where the shape of the array depends on the system. In this work different types of connections inside the S-box are evaluated to find the optimum one.

Segmented tracks

The use of long metal tracks spanning multiple logic blocks was introduced in [50] as *segmented tracks*. It was found to improve speed and reduce delays due to the fact that

applications mapped to the FPGA's functional units tend to require long connections. A similar approach is used in most of Altera's devices. Several works were focused on finding the optimal length and distribution of segments to achieve the best performance in generic applications. Furthermore, the work in [49] [48] proposed a switch box design that is more suited for segmented tracks where unused connections at the end of a segment are removed.

Interconnects in heterogeneous array in Pleiades

In [37] interconnect schemes for heterogeneous arrays are evaluated. The research is focused on interconnects between the coarse block elements in the Pleiades architecture (see review earlier) and tries to overcome the routing problems caused by having blocks with different sizes.

Global interconnects that can connect any part of the array to another were found to be suitable for distant connections, but inefficient for local ones. Furthermore, switching activity of the lines is transmitted for long distances. *Segmented Mesh* architectures improve over global interconnects, but they are difficult to adapt for heterogeneous arrays, as a 2D regular grid has to be found. The proposed solution is to use a *generalised mesh* where wiring channels are used along the sides of each module, with S-boxes on the crossing between the wires, as shown in Figure 3-8.

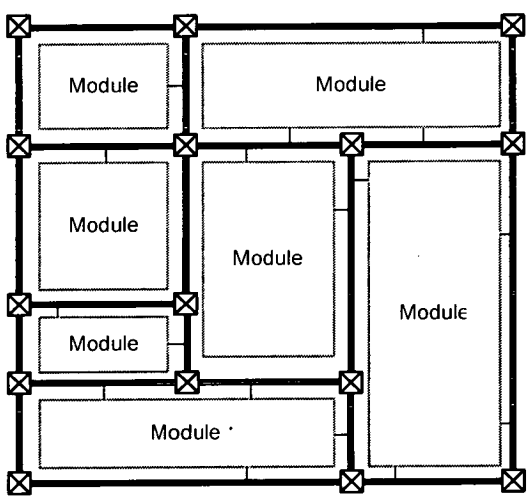


Figure 3-8: Generalised mesh for heterogeneous elements with different sizes in Plaides [37]

The disadvantage is that distant connections go through a lot of switching elements, which introduces delays and might increase the power consumption. Another proposed solution is to use a *hierarchical generalised mesh* with 2 levels of mesh: The elements are grouped into clusters, and an array is made out of clusters. One generalised mesh is responsible for interconnects inside the cluster, and a mesh with larger granularity connects the clusters between each other, as show in Figure 3-9. The tracks are segmented at different levels in the two arrays.

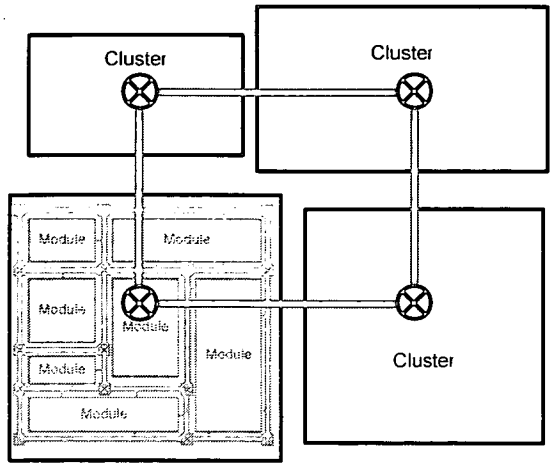


Figure 3-9: Hierarchical generalized mesh in Plaides [37]

2.2.2. Binary interconnect trees

The binary interconnect tree [54] is a useful alternative to the shared bus when cell to cell connections are needed; it uses multiplexers arranged as a tree with each programmable-switch intersection having 3 ports. The advantage of this architecture is that the number of switches used to route the signal grows logarithmically with the distance, which means that the overall delays introduced by the switches are lower. The disadvantage is that this scheme it is not scalable for very high numbers of FUs nor for changes in the number of I/O pins in each cluster.

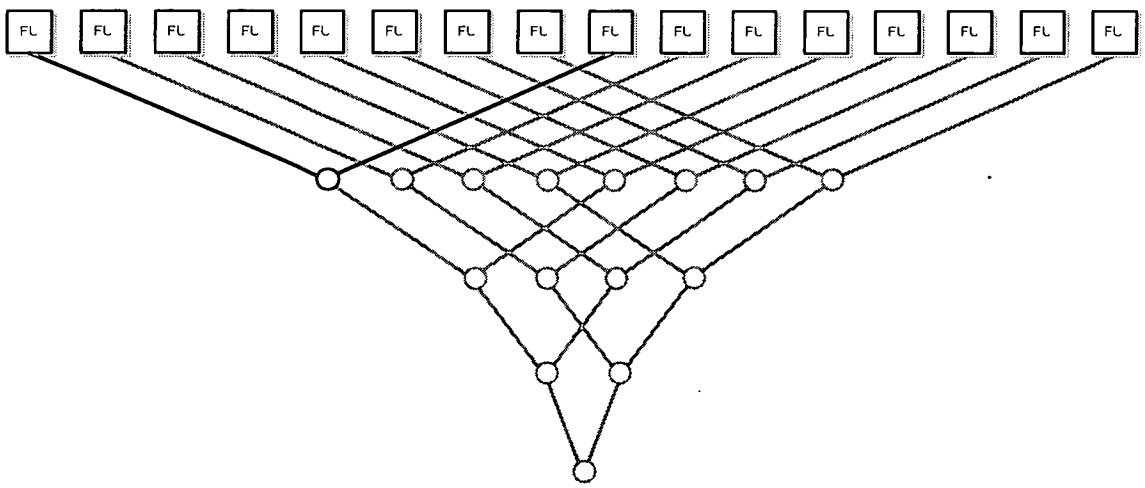


Figure 3-10: Reconfigurable Binary multiplexer-tree interconnect [54]

2.2.3. Hierarchical structures

Hierarchical interconnect structures are useful in applications where data locality is high (neighbouring FUs are making most of the data communication) and only a few signals need to be sent across the chip. Several studies were done on such classes of interconnect and were

found to be efficient for some types of application [55] [56] – in most cases they can improve the speed at the cost of increased area over FPGAs. It should be noted that even though Hierarchical structures and Binary-Trees are conceptually the same in terms of switches, the only difference is the layout and FU-placement used when implementing on silicon.

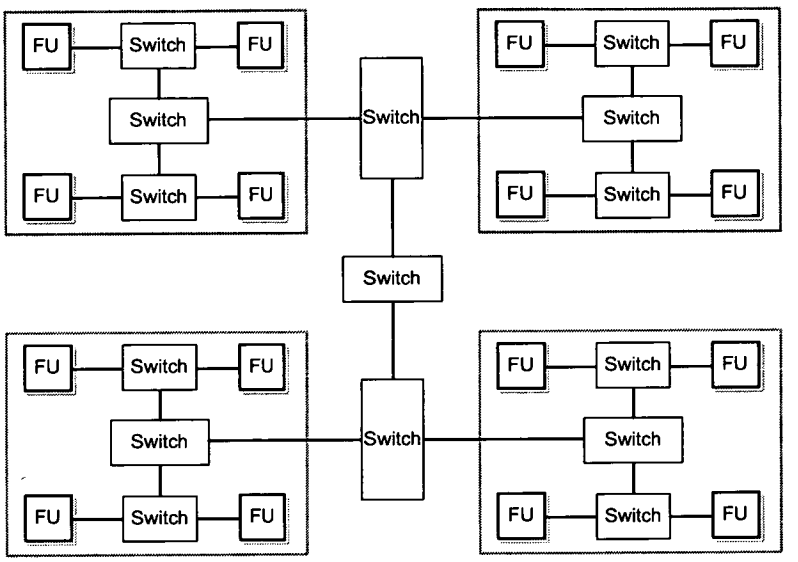


Figure 3-11: Hierarchical FPGA architecture [55] [56]

2.2.4. Combined structures for low-power from LP-FPGA

The power reduction measures in the low-power FPGA from [4] are mainly performed by combining 3 levels of interconnect:

1. Nearest neighbour: High-speed and short lines are present from each functional unit to its 8 neighbour. Very low energy is dissipated in those connections.
2. Mesh Interconnect: Connections between central functional units that cannot be made using nearest-neighbour connections. Those are similar to standard interconnect lines, but the difference is that the number of lines used is lower, and hence less power is dissipated. This is based on a segmented symmetrical mesh.
3. Hierarchical Interconnects: High-delay lines for use between large distant logic blocks on the array. The structure is a mix of a symmetrical mesh and binary-tree architecture with inverse clustering.

Furthermore, to reduce the power consumption of interconnects, circuit techniques are used such as low-voltage drivers on the tracks to reduce the voltage from 1.5V to 0.8V, and hence reduce the power consumed by switching activity.

2.3. Summary

By surveying the existing solution and the on-going work we can identify two gaps:

1. A very large disparity exists between FPGAs and ASICs in terms of cost, power, area, delays and flexibility. This forces applications to chose one of the extremes depending on requirements. This gap needs to be filled with a general solution, or general platform for creating specific solution, as described in Chapter 3, 4 and 5.

2. Amongst the large number of existing couplings of processors and reconfigurable-arrays and the surveyed *reconfigurable computing architectures* there is a lack of a solution that supports high-level programming through C and at the same time addresses critical issues such as low-power and high-flexibility. This is addressed in Chapters 6 and 7.

Chapter 3: Domain-Specific Reconfigurable Arrays:

As described in Chapter 1, there is a need in future portable System-on-Chip designs to achieve a higher computational performance than is currently achieved, while keeping the power consumption at a minimum. Although custom hardwired ASIC designs are currently the choice in such situations, they suffer from a high level of inflexibility and costs not suitable for such rapidly changing requirements and markets. At the same time, programmable solutions such as FPGAs offer flexibility but suffer from high power consumption. Based of the results found in previous work (Chapter 2), the *domain-specific* approach seems to be a promising and extensible solution for achieving a balance between ASICs and FPGAs in order to bridge the gaps in cost and performance between these two alternatives.

The existing domain-specific solutions provide a good cost / performance ratio, however, they are tied to only one application. The main problem with the domain-tailored approach is that it is too time consuming to design a custom datapath from scratch each time a new algorithm/application is encountered in an SoC. Hence, for domain-specific solutions to become useful there is a need to make their creation fast and customisable. A platform and infrastructure to quickly allow the design of such arrays is required, and, to our knowledge, none of the previous works focused on the fast generation of domain-specific architectures. Such customisability is important to allow choosing the exact degree of flexibility required in the architecture according to system-level constraints such as power, area and delays.

The work presented in this chapter can be put in perspective with previous research into domain-specific silicon compilers carried out at the University of Edinburgh; The *FIRST Compiler* [133] generates VLSI designs based on high-level description of computations. This compiler is domain-specific in a sense that it only creates circuits based on bit-serial atomic building-blocks; This greatly narrows the range of applications that can be targeted but gives very high-performance circuits for computations that can be expressed within the scope of the compiler. This compiler can also be coupled with domain-specific standard-cells, as shown in the *SECOND Compiler* [134]. The work presented here takes a similar approach but concentrates on a complete algorithm level rather than one computation, and it also adds the flexibility criteria to the final design.

3.1. Building Domain-Specific Arrays

Ideally the platform for generating domain-specific architectures should be completely automatic, and its only input would be a description of the application using a high-level description language. Another approach is to make the creation of the domain-specific arrays semi-automatic, where the designer would have to manually choose the resources required on the array before it can be automatically created. Even though the semi-automatic methodology gives more responsibility to the designer, it was chosen as a starting target for this work as it allows an easy benchmarking of the performance in the domain-specific arrays. The methodology proposed gives the option to the designer to choose each element of the array from a library of predefined elements. The elements library would be large enough to make it possible to customise the array in terms of functionality and degree of flexibility, which also affect the timing, silicon-area and power consumption. Furthermore, to have a useful platform, the array creation and customisation processes needs to be fast enough to allow testing array with a number of if-then-else scenarios to choose the best compromise between flexibility and performance.

According to the results in the previous work described in Chapter 1, it was decided that an FPGA-like array arrangement and interconnect structure would be best suited for initial

performance testing, as it would allow the reuse of some of the work done on such structures. As described earlier, FPGAs are usually composed of functional clusters (in the FPGA case these are Configurable Logic Blocks) surrounded by programmable interconnects in an island-style fashion to allow connecting the clusters together. Hence, this scheme uses independent elements for routing and for data-processing. If such an arrangement is used for the Domain-Specific Reconfigurable Arrays (DSRAs), which are composed of programmable data-processing clusters and data routing elements, then the *elements-library* would provide different types of interconnects-circuits and operational clusters that would make it possible to generate any array according to the desired functionality and application.

This customisability makes it possible to choose the desired amount of flexibility according to constraints such as performance (i.e. the delays allowed), silicon area and power consumption of the final SoC. The generated array has to fit inside the existing SoC software tool-flow as if it was a standard core. This can be done by generating a pre-routed silicon layout of the array; however the resulting array would not be portable to different fabrication technologies and the array-generation tool would need to know the details of the technology used. This is impractical as only a limited number of processes and fabrication technologies would be supported. The solution used here is to generate the array in a generic synthesisable format so that it can be used as a standard block inside the SoC software tool-flow.

3.2. Proposed reconfigurable System-on-Chip

Since the proposed reconfigurable arrays are domain-specific, in order to perform multiple operations a reconfigurable System-on-Chip would need to contain a number of such arrays each targeting one computation (as shown in Figure 3-1). Usually an array would be created for each computation that needs to be speeded up and all the arrays would run concurrently to achieve a high throughput. The arrangement using a processor and a number of domain-specific arrays in an SoC can also be seen as a compromise between the two existing solutions of using a number of hardwired cores limited to an operation or using a large embedded FPGA that could implement all operations. An SoC bus can be used to provide an easy integration of the arrays with the processors and DSPs, however, a Network-on-Chip (NoC) approach would be more efficient. NoCs are more difficult to implement as currently no standard exists for them. In any approach, the processor would make the synchronisation between the arrays, configure them, provide them with the input data and read back their processed outputs. The array could also have some internal interim buffers, or it could have a Direct-Memory-Access (DMA) to the DSP's memory.

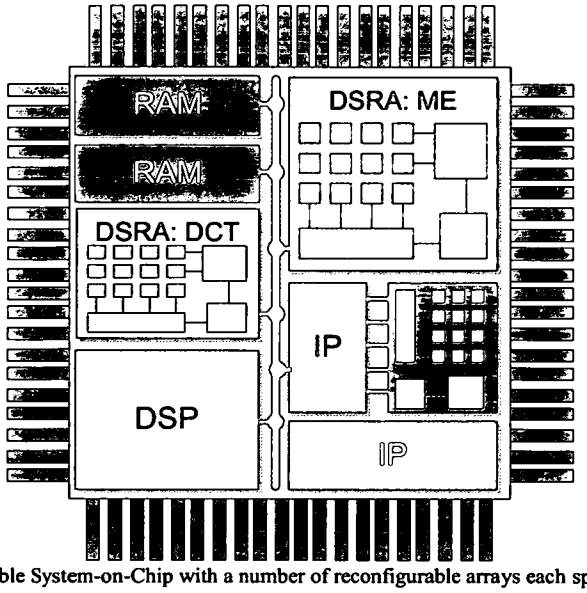


Figure 3-1: Reconfigurable System-on-Chip with a number of reconfigurable arrays each specific to one operation.

3.3. Programmable Clusters

The proposed arrays contain separate elements for data functionality and data routing. The clusters are the main functional elements in the array and they define the operations executable on it. The array was chosen to support heterogeneous clusters, as this can potentially reduce the area and silicon utilisation of the area when compared to a homogenous approach, in case the provided functional units match the required operations. When having a number of different clusters each of them would be responsible for one type of operation. In such a heterogeneous array it becomes possible to add new functionality to the array by augmenting it with new clusters. Individually, a cluster might not be able to perform any practical operations on its own; it is only by connecting several clusters together that a useful computation can be performed; hence, each cluster has I/O pins connectable to other clusters using the programmable switches.

In the proposed scheme, the array is made specific to one domain of application according to the choice of deployed clusters. As will be seen later in Section 4.2, the operation performed by the clusters entirely depends on the application and its requirements in terms of flexibility and performance; typically, each programmable cluster can perform a small set of operations such as add, sub or shift. Clusters usually operate on word-level, e.g. 16-bit or 32-bits. In contrast to generic FPGA architectures, the clusters used here are coarse grain. This reduces the flexibility but improves performance as fewer interconnects are required as was shown in a number of previous architectures.

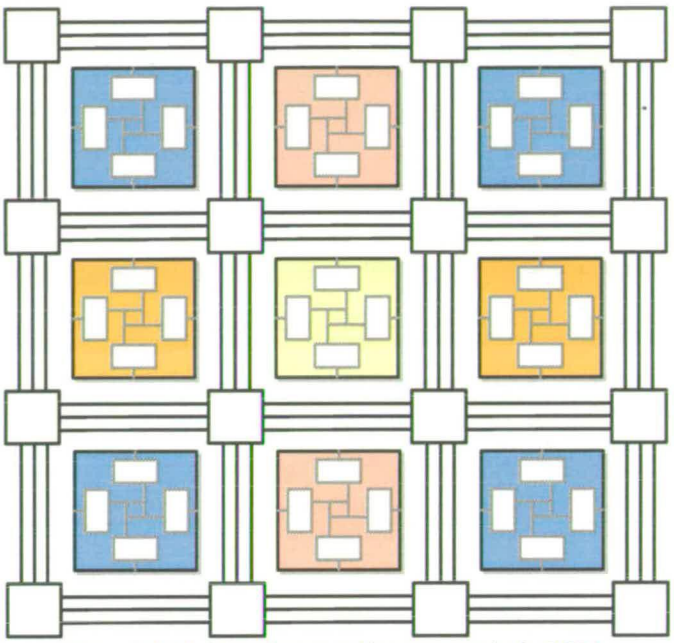


Figure 3-2: Modules, clusters and interconnects in the DSRA

Making the clusters programmable allows the support of different operations or configurations on the same cluster. For example, an ADD cluster could perform additions as well as subtractions. Also, an ADD cluster which was designed as a 32-bit adder can be programmed to perform either a single 32-bit or two 16-bit addition / subtractions. Furthermore, the clusters can be programmable in such a way as to make it possible to select whether they should operate combinatorially or have registered outputs. Such an option can be used to create dynamically customisable pipelines.

Once a number of domain-specific arrays have been generated for a number of applications, the library of clusters described earlier can be compiled. With such a library, an array for any application can be simply created by means of selecting the types, locations and numbers of clusters.

3.4. Interconnects

The role of interconnects is to allow the transfer of data from the output pins of a cluster to the inputs pins of another cluster so that large operational circuits can be formed. Ideally, the switching network would allow the routing of signals between any two cluster-pins in the array at any time. An implementation of such interconnects can be done by using a large multiplexer on each input port of each cluster; this multiplexer would be connected to all the output ports of other clusters and allows choosing the data to route. Although such a multiplexer implementation would be easy to program, it occupies too much area to be economical, and the overhead is not justified since not all the multiplexers would be used at a single time. Hence, there is a need for an interconnect structure that reduces the overhead of

unused programmable-switches while allowing the routing of a wide range of circuits. The programmable switching elements also have to be combinatorial with the minimum delay possible, as opposed to other reconfigurable architecture like PipeRench [29] where the interconnects are registered. In the DSRA interconnects create combinatorial connections between clusters, and any extra implementation details, such as pipelining, would be achieved inside the clusters.

The island-style interconnect scheme used in typical FPGAs fits these requirements, since it provides an area efficient scheme to connect the clusters together, as opposed to the multiplexers scheme. The interconnect mesh uses connection-boxes to connect the cluster's pins to the tracks and switch-boxes to connect the tracks together (see Figure 3-7) to allow sharing the programmable switches between different paths. When using this architecture, extra effort is required to choose the optimum path between two points. Routing techniques have been well developed over the past years and standard routing tools such as VPR [57] can be reused in the DSRA.

Since the clusters are coarse-grain compare to CLBs in FPGAs, the interconnects have to be adapted to the word granularity of the array. Due to the potentially large number of both single-bit and word-wide lines, it was decided that both levels of bit widths have to be supported by using two different levels of interconnect. The word-wide interconnects would be wide enough to efficiently route all widths of signals. As in the examples in Section 4.2, a combination of single-bit and 8-bits tracks can be efficiently used to route signals with widths ranging from 1-bit to 32-bits. When compared to single-bit tracks in FPGAs, using word-wide tracks reduces the number of configuration bits required to route signals, however, the number of routing elements (i.e. multiplexers and programmable switches) stays the same.

In conventional generic FPGAs the configurable switches are implemented as pass-transistors, which allow bidirectional connections between two tracks. To make the generated array synthesisable, the configurable switches have to be implemented using tri-state buffers if bidirectional wires are needed. Tri-state buffers are usually avoided in designs since they may introduce instability in the system. They also increase the area and power consumption of the interconnects when compared to pass-transistors. Using tri-state buffers allows having longer wires since they can support higher loads [52], but such long distances are not really needed in the DSRAs as the data is more local. Two tri-state buffers replacing a bidirectional pass-transistor consume 8 times more area and need 2 configuration bits instead of one, hence the design of the array should try to reduce the overall number of switches needed.

It is also possible to use unidirectional tracks which would make it possible to avoid tri-sate buffers and reduce the overall area of the array, but it comes at the cost of reducing the flexibility of the architecture. The usage of unidirectional tracks depends on the application's requirement; such optimisations are examined in Chapter 5.

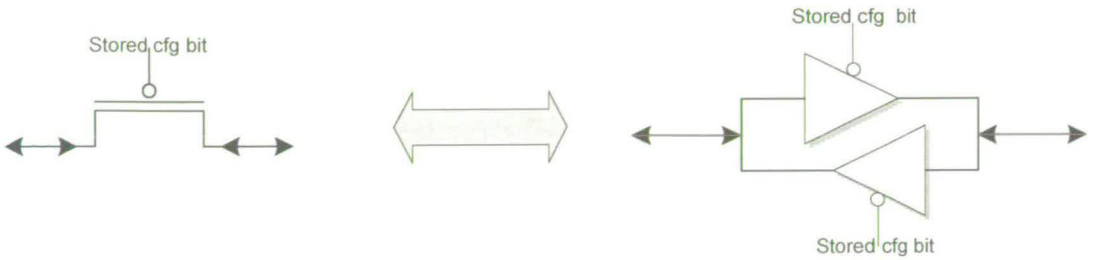


Figure 3-3: Synthesisable equivalent of a bidirectional pass-transistor using 2 tri-state buffers, consuming 8 times more area.

Inline with the remaining elements of the array, interconnects are fully customisable. Parameters include the number of tracks, the width of the word tracks, the flexibility of the connections and switch-boxes. These options affect the flexibility of the array, the routability of designs, the power consumption and area of the final chip; thus they can all be set in accordance with the requirements of the application.

As described later in Section 4.2, the initial sample array was made fully bidirectional and with the maximum flexibility possible in the C-Boxes and the S-Boxes (defined in Chapter 2), as the purpose of this implementation was to measure the initial performance of DSRA. Further optimisations have been later made to the S-Box circuit (Chapter 5).

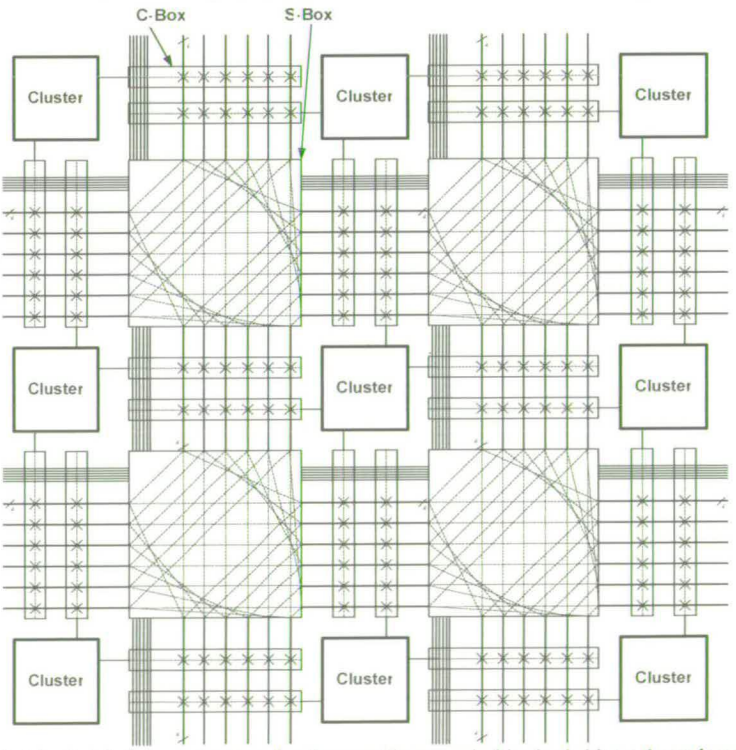


Figure 3-4: Basic island-style interconnect mesh scheme with customisable single bit tracks and word-wide tracks.

3.4.1. C-Box circuit design

Connection boxes allow connecting the pins of the clusters to the tracks. Since the tracks used are bidirectional, the programmable switches between the tracks and the ports have to be

based on tri-state buffers. This is required for the cluster's output pins, as show in Figure 3-5 and Figure 3-6. For the cluster's inputs pins, either a multiplexer or tristate buffers can be use, in order to select which track needs to be routed to the pin. For bidirectional pins, two tri-state buffers have to be used per track. The flexibility measure *Fc* of a C-Box represents the number of tracks the pin can be connected to. For the initial arrays (see next chapter) a high flexibility of *Fc=number of tracks* has been chosen for measuring the initial performance.

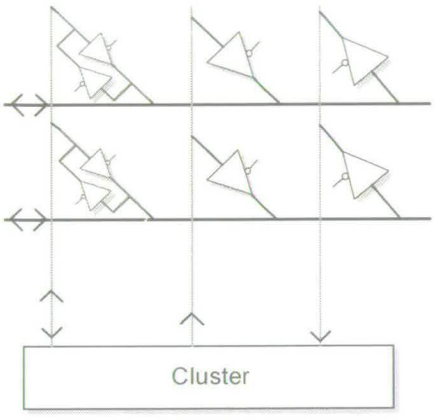


Figure 3-5: Tri-state buffer based C-box

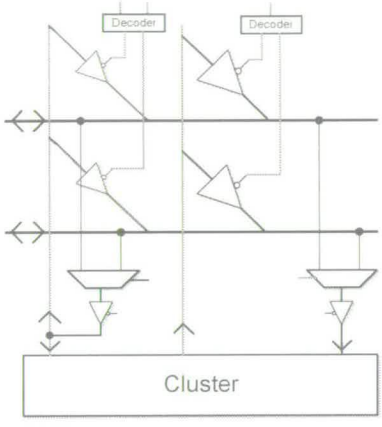


Figure 3-6: C-Box using a multiplexer for input pins only.

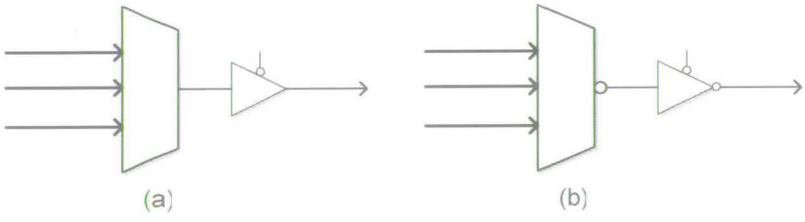


Figure 3-7: Two possible combinations of the MUX and tri-state buffer for use in C-Boxes.

To improve the performance of the interconnect inverting tri-states (or multiplexers) are used, since they have less area, power and delay than the non-inverting ones. This is possible since it is known that each signal between two pins will go through an even number of C-Boxes (in this case 2).

3.4.2. S-Box circuit design

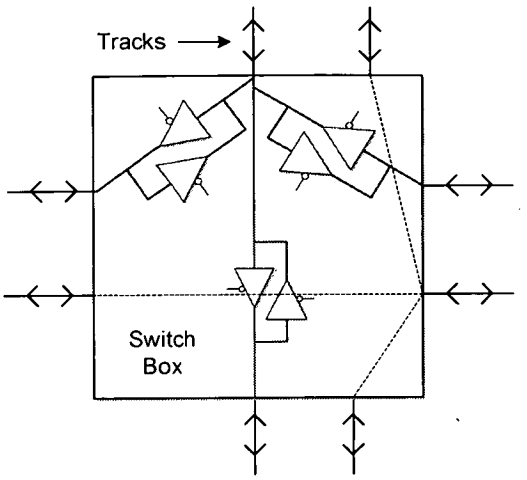


Figure 3-8: S-Box using tri-state buffers

Similarly, tri-state buffers or a multiplexer can be used in the Switch-boxes. This is investigated later in Chapter 5, as such a choice can be application dependent. Unlike the C-Boxes, non-inverting elements have to be used, since a signal can go through an undefined (odd or even) number of S-Boxes to reach its destination. Future examinations can try to use inverting elements while adding a constraint on the routing program to use only an even number of S-box connections.

Again, the initial S-Boxes tested had the highest flexibility of Fs=3, which represents the number of different directions that a signal coming to the S-Box can go to [43]. This value was chosen here for simplicity and can be configured by the designer according to the requirements. The topology used was the *subset* S-Box (see [52] [1]), as this proved useful in FPGA interconnects. Other topologies can affect the characteristics of the array.

3.5. Configuration Memory

The configuration bits controlling the clusters and interconnects have to be stored in a memory device. The configuration memory contains the settings of all the configurable switches and multiplexers in the array. This includes the settings of all the clusters as well as the connection- and switch-boxes. Each cluster and its surrounding C-boxes require in the order of 100-200 bits of memory. An S-Box needs around 250 bits. The large number of configuration bits required is due to the high flexibility of the C- and S-boxes. Reducing this flexibility will reduce the required memory and the area of the array.

3.5.1. Requirements and observations

The memory needed to store the configuration has the following characteristics which are described below:

- Read latency is unimportant, as no data will change quickly; this actually depends on the rate of reconfigurability, however, it would never require changing the configuration in a single clock cycle – for the testing purposes at least.
- The time taken to write to the memory is not crucial, as it again depends on the rate of reconfiguration (see below).
- The data will not be read from the memory (except if debug capabilities are needed); hence each bit-cell can have its output connected to the configurable switch.
- All outputs need to be available at all times.
- The memory should be spread around the chip, since the memory cells should be kept next to the switches and clusters to minimise wires lengths.

The rate of reconfiguration of the array is entirely dependent on the application. It could be measured in months, in case the reconfiguration is only part of a firmware update or functionality change, or it could be in fractions of a second if the application needs to dynamically change the behaviour of the array according to external changes. Thus several types of memory elements, such as non-volatile flash or SRAM can be used according to the requirements.

However, the fact that the array is required to be portable to different processes and fabrication technologies limits this choice. Flash or SRAM memory cells, as the ones used in FPGAs, are not synthesisable. Stable synthesisable memory is restricted to flip-flops and latches. In the configuration memory for DSRAs, all the bits of the memory-cells have to be available all the time to constantly control the multiplexers and switches. Thus, a standard SRAM memory block as the ones provided by foundries such as UMC, might not be suitable as a configuration memory, since in usual SRAM block only the output-bits of the currently selected row are available at one time. To use SRAM technology, the definition of a single-bit SRAM cell and a controller would be needed, which requires circuit level and foundry specific designs. Hence, a synthesisable latch or register based memory is more appropriate.

As with the bidirectional tri-state switches, the use of flip-flops as configuration memory increases the area needed per configuration-bit by around 2.7 times when compared to SRAM-cells. Hence, the overall number of configuration bits and programmable switches used (or saved) in the array has a significant impact on the total chip area.

To facilitate dynamic reconfiguration of the array, it should be possible to partially change a small data-block in the configuration memory at run time. The data change should only affect its associated hardware and not the configured circuit for the rest of the array.

The easiest option for the configuration memory would be to use registers arranged as shift-registers. The output of each register is connected to the multiplexer or switch it controls. The programming of the registers can be done in a bit-serial manner by filling the shift register with the configuration bit-stream. Each cluster and its corresponding c-boxes can be grouped

together and a wide shift-register is assigned to it. The block would have one *bit-input* and one *bit-output* pins for configuration. Multiple blocks can be cascaded by connecting the *bit-out* of the current block to *bit-in* of the next block, hence a number of blocks can be configured serially, as shown in Figure 3-9

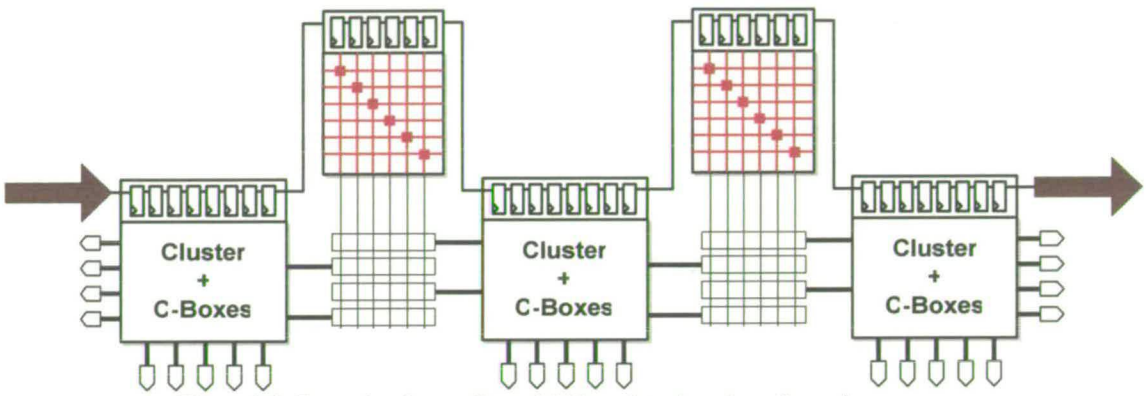


Figure 3-9: Example of cascading of shift-register based configuration memory.

In the extreme case, the configuration shift-registers of the whole array can be cascaded so that the array can be configured by a single bit-stream. However, to enable quick dynamic reconfiguration, the array needs to be split in small regions each region requiring a separate configuration bit-stream input. In the initial design it was decided that every row of the array has one input bit for configuration.

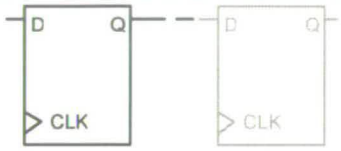
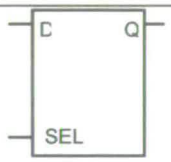
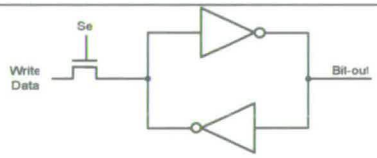
3.5.2. Alternatives and improvements to shift-registers

In typical FPGAs, very high current is drawn by the chip during the configuration process as all the programmable elements would be switching on and off while loading the configuration bitstream. According to the rate of reconfiguration, this exhibited power can become an important factor. As described above, flip-flops arranged as a shift-register are quite simple to operate. However, the configuration bits would have to hop between different registers, triggering their programmable elements unnecessarily before arriving to its target flip-flop. To avoid this needless switching activity, an extra enable signal can be used so that the output of the flip-flops is disabled during the writing.

The other alternative to flip-flop memory cells is latches. As seen in Table 3-1, the area of a latch is around 60% that of a register. However, the multitude of latches cannot be simply cascaded into shift-register and require a controller to select which individual bit to program, which adds an extra area overhead. Such a controller has been tested and designed to allow addressing every programmable block (i.e. S-Boxes and clusters with their associated C-boxes) individually. The controller accepts input configuration data and target block address. Since the writing occurs in a word-serial manner, the width of the data line affects the speed of writing and the number of decoders needed for the latches circuit (the performance

measure below uses widths 1, 4, 8 and 16 bits). On the other hand, the width of the addressing line for the controller depends on the number of clusters in the array. Also, internally the controller would need to count which word of the configuration bitstream is being received so that it can be sent to the correct latches. Since this counting scheme would affect the power consumption it was decided to compare both grey-counters and one-hot counters.

Table 3-1: Area comparison of configuration memory cells.

		Minimum area in 0.18μm technology
Register		81 μm^2 /bit
Latch		48 μm^2/bit without controller
SRAM cell (for comparison)		22 μm^2/bit without controller

The results of the area of the configuration memory (along with the corresponding controller) and the configuration power are shown in Table 3-2. The results shown are for programming a row of clusters having around 650 configuration bits. It should be noted that the area is for UMC 0.18μm technology and the power consumption is that consumed if all the writing was done at the same speed. By comparing flip-flop implementations 1 with 2 we can see that adding a signal to disable the configuration while programming results in a 33% power reduction at the cost of 8% increase in total area. For the latches, this is not the same, as seen for cases 5 and 8, since the power increases in 8 slightly by 5% (while areas also increases by 10%).

Table 3-2: Area and power of different control circuit and configuration memories

Implementation	Routed Area (μm^2)	Configuration power (μW)
1-FF arranged as shift-register	52,867	488
2-FF, arranged as shift-register, disable while reconf.	57,135	323
3-Latch, grey counter, 1 bit / cycle	49,306	151
4-Latch, grey counter, 4 bits / cycle	43,568	96
5-Latch, grey counter, 8 bits / cycle	42,324	104
6-Latch, grey counter, 16 bits / cycle	41,824	154
7-Latch, grey counter, disable while reconf., 8 bits / cycle	46,919	110
8-Latch, one-hot counter, 8 bits per cycle	45,629	133

When comparing implementations 3, 4, 5 and 6 containing latches with grey-counter based controllers, we can see that the best power/area performance is achieved for implementations *4* and *5* based on 4 and 8 bits word-wide data. Also we can see that the one-hot counter based controller does not offer any advantages over the grey-code one, as it consumes more power and occupies more area. It can be clearly seen that latches based memory is superior to the flip-flop based one, as it consumes up to 70% less power and 23% less area (implementations 4 and 2). However, it should be noted that a shift-register implementation easily allows the configuration data to be read back from the array, while the controller for the latch based one does not allow this. Such a feature can be useful to verify the programming in applications like fault-tolerant circuits.

3.5.3. Further improvements

Several techniques that are employed in existing reconfigurable systems for improving the performance of the configuration memory can be used in the proposed architecture. For example, fast dynamic reconfiguration can be enabled like in DP-FPGA (See Chapter 2) by using a large RAM that temporarily stores a number of configuration-bits. The processor could send multiple configuration bit-streams in parallel to the RAM and then one configuration can be uploaded to the array. The transfer of the configuration from RAM to the array occurs much faster than if the configuration was sent serially from the processor to the array directly. With the RAM storing multiple configurations, a dynamic switch between configurations can be made quickly and efficiently without much data transfer between the processor and the array. Furthermore, the processor is free during the reconfiguration from the RAM, and hence it can be used to execute other computations.

In reconfigurable architectures like *Xilinx Virtex 4* [[1]] it is possible to reuse the configuration registers as general purpose variable shift-register. In our array, it would be possible to make the shift-registers of unused blocks configured to be used in the application. However, several issues have to be solved, like having special configuration bits that sets whether the configuration shift-register of the block is used or not and having c-boxes to connect the configuration *bit-in* and configuration *bit-out* of the block to the routing tracks. Another issue would be to make the size of the shift-register programmable and to be able to read the value at each register.

3.6. Design-Tools flow

In contrast to embedded FPGAs, the proposed domain-specific reconfigurable arrays are integrated with the SoC as a normal core since the DSRAs are provided as synthesisable code. However, the use of these reconfigurable cores adds extra steps to the design-flow as shown in Figure 3-10. The arrays are designed in such as way that the overall SoC design-flow is kept the same and only a small number of new tools is used. The new steps are described below for the design-entry, verification and implementation stages.

3.6.1. Design entry and array generation

As with standard SoC system, early in the design stages of the system a vague partitioning between hardware and software implementations can be achieved by identifying the compute intensive computations of the target application. Regardless of the flexibility required in these computations, they can be implemented efficiently on a reconfigurable array with the cost of an added area overhead to the chip. Hence, depending on the area constraints a decision has to be made on the algorithms to target, the number of arrays to be used and the flexibility of each array. Since the arrays provide a flexibility margin, the initial partitioning can be modified later in the design.

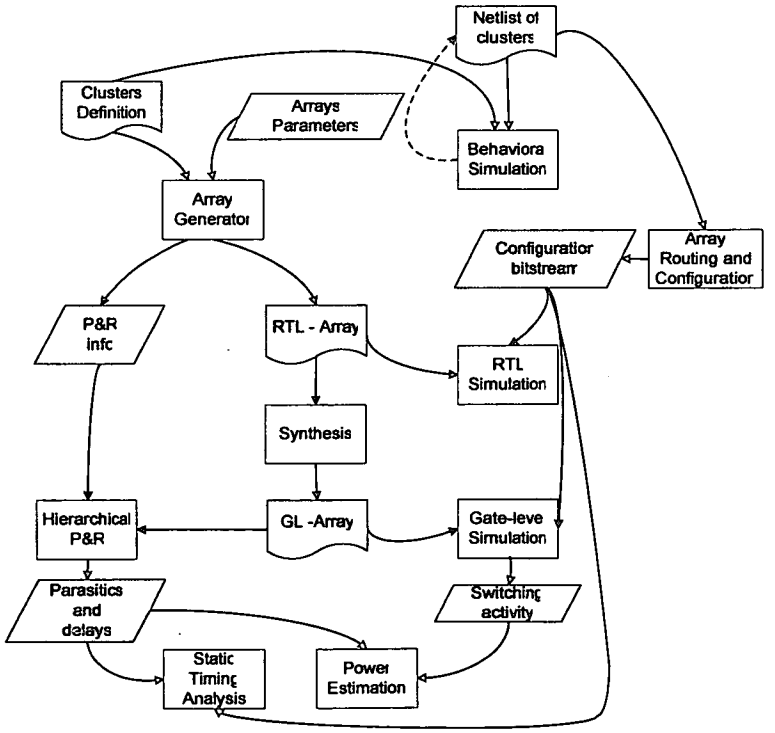


Figure 3-10: System-on-Chip design-flow when using synthesizable reconfigurable arrays.

The programmable clusters used in the array define the application of the array and its flexibility. The clusters can be chosen from an existing library or defined as synthesisable HDL by the designer. The use of a library of clusters improves design-reuse and reduces the design time. To correctly design the clusters, the algorithm has to be analyzed and the basic operations extracted. Another approach to the cluster design is to analyse the existing hardware implementations of the algorithm and identify the common basic operators; designing the clusters to support all the possible implementations allows controlling the flexibility of the clusters.

Table 3-3: Options given to array generation tool

Number of rows, columns
HDL definition of clusters
Position of each routable pin (North, South, East, West)
Placement and number of each type of cluster
Type of Interconnects
Number of bit-wide and word-wide Tracks

The heterogeneous array of clusters is generated automatically from the clusters definition. A tool was developed to read and analyse a Verilog HDL code defining the clusters in order to generate the required connection-boxes and switch-boxes around the clusters. The array generation program is given the parameters of the required array, such as its size, the cluster's arrangement inside it, the locations of the pins on the cluster, the number of tracks and the type of interconnects (as shown in Table 3-3 and Figure 3-11). The array is generated as a synthesisable RTL code.

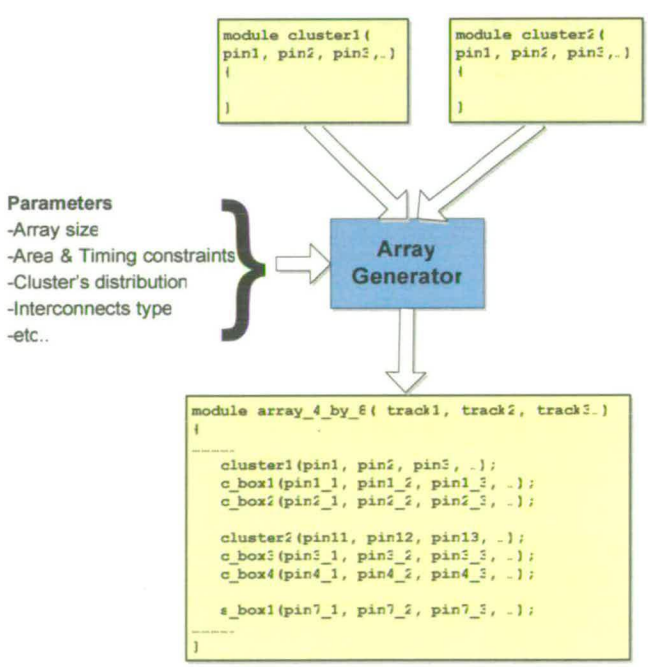


Figure 3-11: Inputs and outputs of the array generator

3.6.2. Array programming and testing

Mapping a design to an array is done manually by writing an HDL netlist of interconnected and programmed clusters. This task is simple since a useful datapath is usually built using a dozen of clusters; the number of clusters in typical circuits does not exceed 64, which does not lead to a large netlist. The designer needs only to connect the clusters together, since the configuration of the switch-boxes and connection-boxes is done automatically, as described in the next section. The placement of the module, i.e. the choice of which physical cluster to use if more than one clusters of the required type exists, is also done manually.

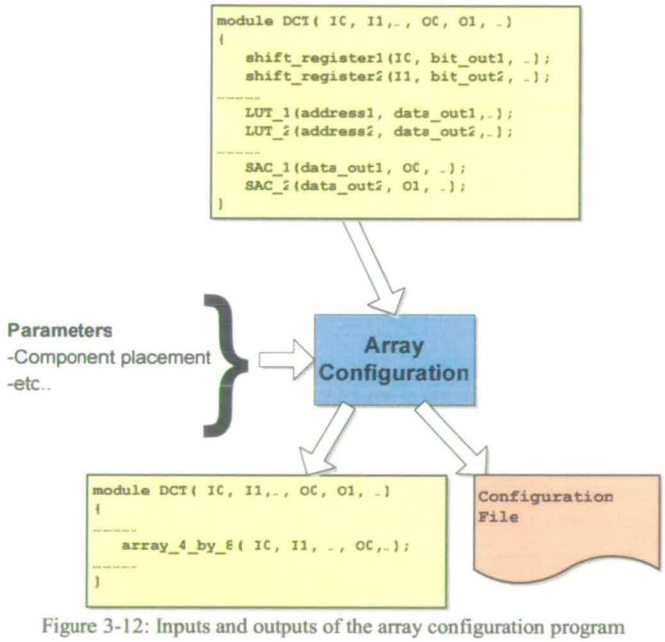


Figure 3-12: Inputs and outputs of the array configuration program

The routing program, which is based on the routing engine provided in VPR [57], generates the required configuration of the connection-boxes and switch-boxes to correctly map the netlist to the array. VPR was modified to allow it to create a configuration bitstream for the interconnects in the array to build the input circuit. This bitstream is then used to configure the array in order to establish. VPR was also augmented with the ability to generate the configuration bits as scripts usable at the different stages of the design, like HDL scripts to test the configured array (both at RTL and gate levels) and scripts for timing-analysis of the mapped configuration (e.g. using PrineTime from Synopsys). The original VPR was also limited to homogenous CLBs and has been modified to support heterogeneous clusters that can each have a different number of I/O ports.

Table 3-4: Example of mapping a DCT computation to the arrays

```
module one_d_idct_seq_elements(I0, I1, I2, I3, I4, I5, I6, I7,
        O0, O1, O2, O3, O4, O5, O6, O7,
        clk, rst,
        load_sregs, en_sregs, add_sub, clr_sac );

        input           clk, rst;
        input  [11:0]   I0, I1, I2, I3, I4, I5, I6, I7;
        output [11:0]   O0, O1, O2, O3, O4, O5, O6, O7;
        input            load_sregs, en_sregs, add_sub, clr_sac;
                wire    [7:0]    d0, d1, d2, d3, d4, d5, d6, d7;              // Output of ROMs
        wire             data_sr0, data_sr1, data_sr2, data_sr3;
        wire            data_sr4, data_sr5, data_sr6, data_sr7; // Output of shift-reg
        wire   [11:0]   I0_a, I1_a, I2_a, I3_a, I0_s, I1_s, I2_s, I3_s;

        add_sub_12b add1 (clk, rst, 1'b0, I0, I7, I0_a);
        add_sub_12b add2 (clk, rst, 1'b0, I1, I6, I1_a);
        add_sub_12b add3 (clk, rst, 1'b0, I2, I5, I2_a);
        add_sub_12b add4 (clk, rst, 1'b0, I3, I4, I3_a);
        add_sub_12b sub1 (clk, rst, 1'b1, I0, I7, I0_s);
        add_sub_12b sub2 (clk, rst, 1'b1, I1, I6, I1_s);
        add_sub_12b sub3 (clk, rst, 1'b1, I2, I5, I2_s);
        add_sub_12b sub4 (clk, rst, 1'b1, I3, I4, I3_s);
        // ROMs, output is 8-bits
        coef_odd_even_rom0 lut0 (d0, {data_sr6, data_sr4, data_sr2, data_sr0});
        coef_odd_even_rom2 lut2 (d2, {data_sr6, data_sr4, data_sr2, data_sr0});
        coef_odd_even_rom4 lut4 (d4, {data_sr6, data_sr4, data_sr2, data_sr0});
        coef_odd_even_rom6 lut6 (d6, {data_sr6, data_sr4, data_sr2, data_sr0});
        coef_odd_even_rom1 lut1 (d1, {data_sr7, data_sr5, data_sr3, data_sr1});
        coef_odd_even_rom3 lut3 (d3, {data_sr7, data_sr5, data_sr3, data_sr1});
        coef_odd_even_rom5 lut5 (d5, {data_sr7, data_sr5, data_sr3, data_sr1});
        coef_odd_even_rom7 lut7 (d7, {data_sr7, data_sr5, data_sr3, data_sr1});
        // Input Shift-registers
        sr_12b in_sr0(clk, rst, I0_a, data_sr0, load_sregs, en_sregs );
        sr_12b in_sr2(clk, rst, I1_a, data_sr2, load_sregs, en_sregs );
        sr_12b in_sr4(clk, rst, I2_a, data_sr4, load_sregs, en_sregs );
        sr_12b in_sr6(clk, rst, I3_a, data_sr6, load_sregs, en_sregs );
        sr_12b in_sr1(clk, rst, I0_s, data_sr1, load_sregs, en_sregs );
        sr_12b in_sr3(clk, rst, I1_s, data_sr3, load_sregs, en_sregs );
        sr_12b in_sr5(clk, rst, I2_s, data_sr5, load_sregs, en_sregs );
        sr_12b in_sr7(clk, rst, I3_s, data_sr7, load_sregs, en_sregs );

        sac_16b sac0(clk, rst, d0, O0, add_sub, en_sregs, clr_sac);
        sac_16b sac1(clk, rst, d1, O1, add_sub, en_sregs, clr_sac);
        sac_16b sac2(clk, rst, d2, O2, add_sub, en_sregs, clr_sac);
        sac_16b sac3(clk, rst, d3, O3, add_sub, en_sregs, clr_sac);
        sac_16b sac4(clk, rst, d4, O4, add_sub, en_sregs, clr_sac);
        sac_16b sac5(clk, rst, d5, O5, add_sub, en_sregs, clr_sac);
        sac_16b sac6(clk, rst, d6, O6, add_sub, en_sregs, clr_sac);
        sac_16b sac7(clk, rst, d7, O7, add_sub, en_sregs, clr_sac);
endmodule
```

3.6.3. Verification

Three levels of simulations can be achieved with the synthesisable arrays: Behavioral, RTL and Gate-level. With the HDL definitions of the clusters and the design to be mapped to the array in netlist format an early behavioural simulation can be used to verify and debug the functionality of the netlist of clusters.

This netlist is then passed to the VPR-based *routing* program along with the placement information that describes where each cluster is placed on the array. The configuration bits generated after routing can be loaded onto the array for simulation of the validity of the routing both at RTL and gate level definitions of the array. Similarly, the configuration bits for the array can be used to perform accurate timing analysis that depends on the configuration loaded on the array. The gate-level simulation is useful to make estimation of power consumption.

It should be noted that the verification, performance evaluation and analysis processes are done using the existing SoC tools, unlike commercial embedded FPGA architecture where new tools need to be used. Furthermore, the synthesisable reconfigurable array does not require extra design domains such as mixed-mode design; another advantage is that the verification process can include the whole integrated SoC for accurate simulation, unlike embedded hard-cores.

3.6.4. Implementation

The array is implemented as any soft-core with typical synthesis, placement and routing software. Better performance is achieved if the synthesis of the elements of the array and their placement and routing is performed using a hierarchical methodology. The array generation program outputs guideline files for the place and route software to efficiently perform floorplaning and routing of tracks. The same hierarchical methodology is used to implement the full SoC design. Having a routed SoC allows the extraction of typical parasitic and delay data for the array which permits having an accurate timing and power estimations of the SoC; this also allows comparing the performance of different scenarios and configurations for the array, which helps evaluate the overhead consumed by the added flexibility.

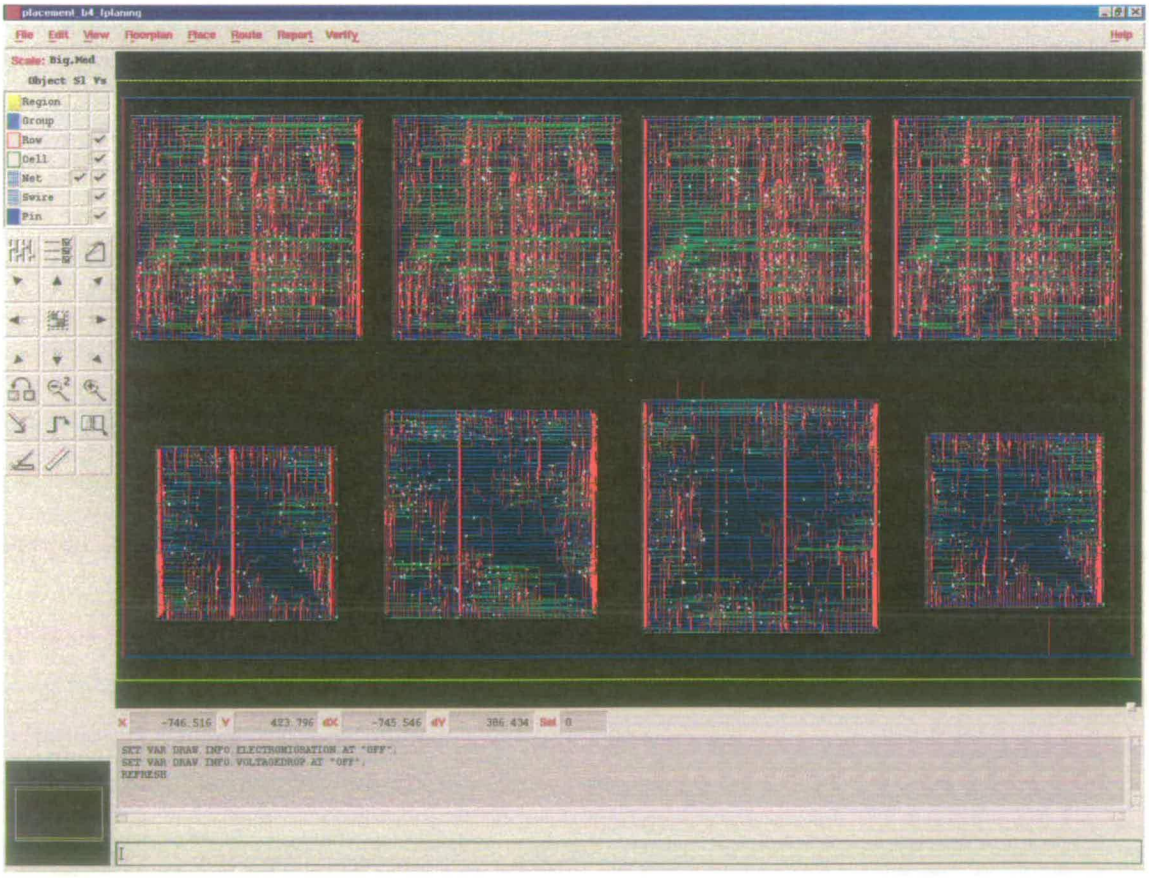


Figure 3-13: Example of placed and routed arrays using *Cadence Silicon Ensemble*.

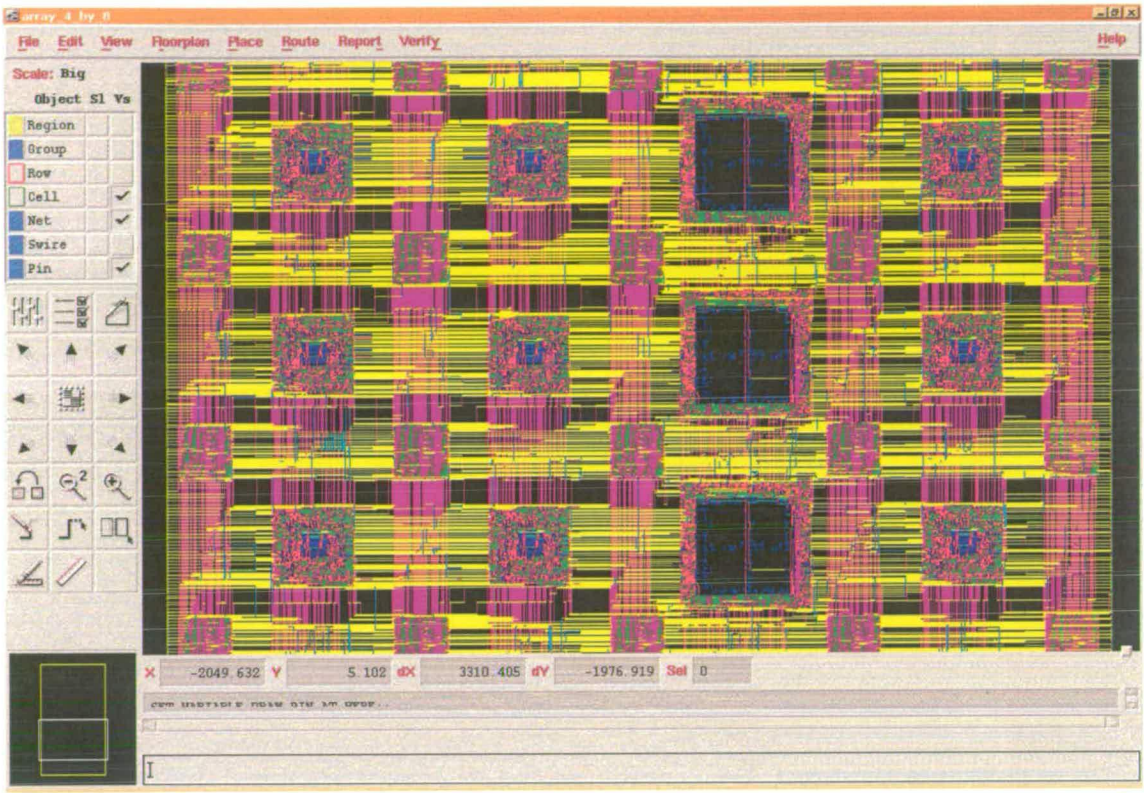


Figure 3-14: Example of placed and routed arrays using *Cadence Silicon Ensemble* showing the interconnect wires.

3.7. Problems and future work

As can be seen in Figure 3-14, one potential problem is the fact that different clusters can have different sizes, which might lead to wasted silicon area. To overcome this, the designer has to ensure that all the clusters and their associated C-Boxes have a similar height and width. If this is not possible, large clusters can be split into two smaller ones, or it can also be floorplanned in a rectangular shape to reduce the wasted area.

If the proposed architecture proves to provide good performance benefits, then a future improvement would be to allow automatic mapping of applications to the array. This can be done from an HDL definition of a circuit where a synthesiser would convert it to the coarse-grain clusters. Ideally, such an operation would also be done from a higher description level like C/C++.

3.8. Conclusion

The architecture introduced uses heterogeneous coarse-grain clusters with an interconnect structure similar to that used in commercial FPGAs. Also, the proposed methodology integrates well with existing SoC tool-flows. In order to create a DSRA targeting a new application, the designer has to identify the repetitive basic operations in the algorithm and create a programmable *cluster* in HDL to provide that operation. Eventually, once a number

of DSRAs have been designed for several applications a library of clusters can be built; at such a stage, creating an array for a new application becomes as simple and time-effective as choosing the clusters from the library. The array generator uses the HDL definitions of the cells and creates the appropriate DSRA. The designer can customise the type of interconnect used, the positions and number of the clusters as well as the locations of the pins of each cluster. Since the generated arrays are synthesisable, this software flow fits well with the existing SoC design tools.

Programming the DSRA takes the same effort as typical ASIC design: The design to be mapped has to be written as a netlist of connected clusters before a configuration can be generated for the array. Similar to FPGAs, automatic routing tools are used to hide the interconnect infrastructure from the designer to simplify programming. The performance of sample DSRA arrays generated using the proposed technique is presented in the next chapter.

Chapter 4:

Domain-specific reconfigurable array for video coding

The main applications that would immediately benefit from reduced power, increased throughput and increased flexibility are audio and video applications as well as implementations of Software Defined Radios (SDRs). Standards such as MPEG-4, H.263 and H.264 contain complex video algorithms such as Motion Estimation and DCT that require a high data throughput. Current implementations of these algorithms on DSPs need a high operating frequency and hence consume a high power. A dedicated ASIC hardware solution is not appropriate for such applications, as these standards keep changing and a re-spin of the chip is not cost-effective. Thus, such algorithms represent a good target for the use of domain-specific reconfigurable arrays. The use of DSRAs for these applications should provide enough flexibility to support a number of implementations while at the same time they should offer a lower area and power consumption than FPGAs. To measure this, experimental arrays were designed for the two main computationally intensive parts of low-profile MPEG-4 encoding: *Motion Estimation* and the *Discrete Cosine Transform*.

The two arrays are only sample unoptimised arrays to help prove the concept of DSRAs and to measure any potential performance improvements over DSPs and FPGAs. It should noted that the proposed framework provides a generic solution, even though these chosen examples are specific applications. the The array design and evaluation process includes first the analysis of the target algorithm to identify the required operations, and then the creation of clusters, which can be also composed of subclusters to perform the basic operations of the application. These clusters are then combined together through reconfigurable interconnects. To measure the performance of a generated DSRA, benchmarks are mapped to the clusters making the array and the performance is compared to other technologies such as FPGA and ASIC.

4.1. Overview of the targeted MPEG operations

In MPEG video, the moving images are composed of consecutive frames. Each colour image is composed of 3 elements: The luminance (Y) and two chrominance (C_B and C_R) parts. The images are divided into small 16x16 pixels blocks. Each block consists of one 8x8 C_B pixels blocks, one 8x8 C_R pixels block and four 8x8 Y pixels blocks (which can be considered as one large 16x16 Y pixels block).

The general structure for a frame encoder and decoder is shown in Figure 4-1. The encoder computes the *motion information* and *texture information*. These data are multiplexed to form the compressed bitstream; using which the decoder is able to reconstruct the frame. In MPEG-4, the actual compression of video data is done at 3 different levels:

- *Motion Estimation* (ME) is used to reduce temporal redundancy in the image sequence, as the consecutive frames of a video sequence tend to be highly correlated. Hence the *motion information* contains the movement data between the current frame and the previous frame.
- Transform-domain coding, here Discrete Cosine Transform (DCT), and quantisation are used to reduce the spatial redundancy found in a single frame.
- Finally, Bitstream compression is used to compress further the generated data.

Motion Compensation (MC) is the operation of reconstructing a frame from a previously constructed frame knowing the motion information. This is used at the decoder to reconstruct the video. However, as shown in Figure 4-1, the encoder also requires this operation so that it knows the previous reconstructed frame that the decoder is using. The decoder needs only to know the motion information and the error between two pixel-blocks in order to reconstruct the current block, and hence the full frame.

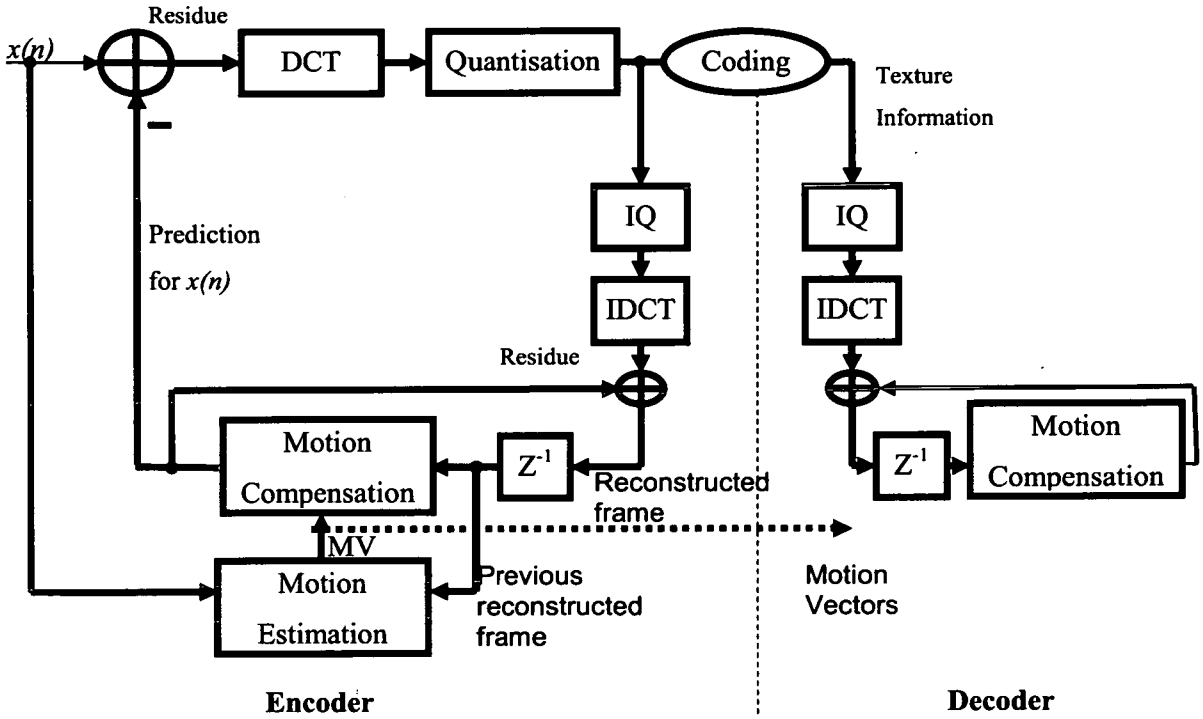


Figure 4-1: Block Diagram of operations in Encoder and Decoder for rectangular objects from [130].

The MPEG-4 standard only specifies how the MPEG bitstream data needs to be formatted and how the decoder should use the information contained in the bitstream. The standard leaves the choice open for the algorithms used to make specific computations, hence the existence of multiple coding algorithms with different characteristics in terms performance and cost.

4.2. DSRA for Motion Estimation

4.2.1. Algorithm

Motion Estimation (ME) is the process of matching the current block to be coded (in the current frame) with a similar block from the previous frame. As video sequences tend to be highly correlated, it is easier to transmit the movement of a block between 2 frames rather than transmitting the completely coded block.

In general a ME algorithm uses a cost criterion to compare the current block to some blocks in the previous frame (limited within a search area) and selects the best suited one where the error between the two blocks is the smallest. This is shown in Figure 4-2, where an area is searched for an *NxN* block matching the block in the current frame. The Motion Vector (MV) represents the 2D movement vector between the current block and the most suitable previous block found.

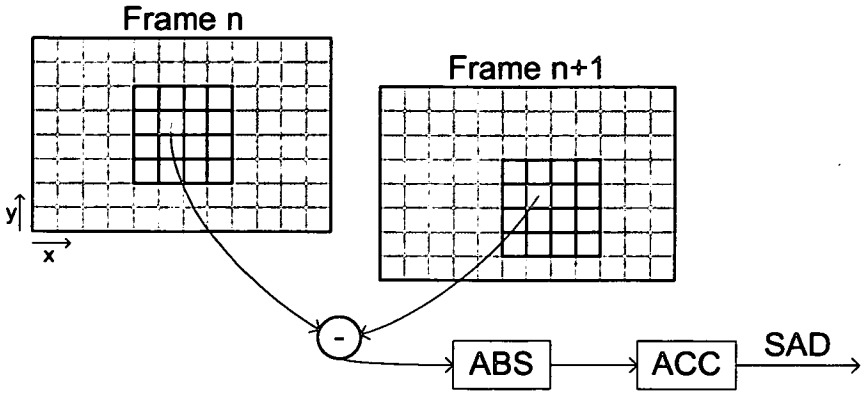


Figure 4-2: Block-matching between current and previous frames.

A criterion function suitable for finding the best motion vector is the sum of Mean Squared Error (MSE) of all the pixels of the two blocks compared. However, to reduce the computational needs nearly all algorithms use the Sum of Absolute Difference (SAD) function. The SAD between two blocks is the sum of absolute differences between pixels from the current block and their corresponding pixels in the previous block. For a MV of coordinates (x,y) the SAD is:

$$SAD_N(x,y) = \sum_i^N \sum_j^N \left| original(i,j) - previous(i+x, j+y) \right| \qquad (4.1)$$

Where N is the size of the block (which could be 8, 16 or 32).

A number of motion estimation algorithms exist based on the SAD calculation and differ by the order, number and size of blocks compared as well as by the bit-width of the pixels. The basic ME uses the Full Search Block Matching Algorithm (FSBMA, in which the SADs for all the possible blocks in the search area are calculated and the motion vector giving the minimum SAD is selected. This gives the best results and has a simple structure when implemented. However, the FSBMA consumes a long computational time when compared to other algorithms. If *NxN* is the size of the block and *(N+P+Q)x(N+P+Q)* the size of the search area, then there are *(P+Q+1)x(P+Q+1)* candidate blocks to be tested. The loop needed for the calculation of the motion vector for only one block is:

```
For m = -p top
   For n = -p to p
      For k = 1 to N
         For l = 1 to N
            SAD(m,n) = SAD(m,n) + | x(k,l) - y(k+m, j+n) |
         End l
      End k
      If SAD < SADmin
         SADmin = SAD
         MV = (m, n)
      End if
   End n
End m
```

Most of the existing algorithms for speeding-up the computation are based on reducing the number of tested motion vectors. One such popular algorithm is the Three Step Search (TSS) [60] where the first step of the search evaluates 9 uniformly located candidate points and selects a winner with minimum SAD. In the second step, the search is refined at the area around the winner of the previous step. Again, 9 candidates are evaluated, but this time the distance between candidates is halved. Finally, in the third step the 9 blocks around the winner in step 2 are evaluated and a final motion vector is chosen. A large number of other algorithms exists tho reduce the number of tested points further, usually at the cost of a quality degradation; for example: The New TTS [61], Fast TTS [62], Diamond Search [63], Spiral Search [64] (where the search moves spirally around the vector predictor location till a threshold is passed, thus having a dynamically changing search area), M-IBOS [65], 2SMWS [66] and hierarchical Search. Another technique to speedup the blocks comparison is to change the tested blocks themselves, such as using size-downsampled blocks of 8x8 or 4x4 instead of 16x16 [67] or bit-downsampled of 4-bits or 2-bits instead of 16-bit [68].

4.2.2. Existing reconfigurable architectures

An architecture targeting ME with flexibility would ideally support all the search algorithms listed earlier. Pervious work on motion estimation has lead to architectures providing flexibility in the supported algorithms, however, it is very limited and not adequate to allow changing between different coding standards. E.g., the hardwired elements proposed in [69] can be configured at run-time to support 3 different bit-widths to save power; however, only one basic algorithm is supported. Similarly, [70] and [71] present architectures supporting only one algorithm but having flexibility in the size of blocks and search area. The hardware in [72] and [73] offer reconfigurable elements that can switch between two algorithms differing by the number and the order of blocks searched.

Processor solution

Most previous flexible solutions for implementing ME are based on processors; in such solutions the processor supports specific instructions that help in rapidly performing the ME computation. This includes instructions such as absolute-difference calculation and instructions for *min* and *max* calculation as in [74]. The absolute-accumulate instruction is sometimes provided [75] to allow an easier calculation of the total SAD.

Another method for improving a processor's performance in video applications that has a benefit to ME is the increase of data parallelism: In [76] sub-word parallelism allows the execution of four 16-bit operations on a 64-bit datapath simultaneously. This same Single Instruction Multiple Data (SIMD) concept is used in the multimedia tailored ARMv6 architecture [77] which performs four 8-bit SAD calculations in one cycle. This reduces the total processing time of 4 pixels down to 3 cycles.

Non-reconfigurable array structures for FSBMA

Basic systolic-array architectures for motion estimation have been presented in [82] and [78]. A large number of newer architectures are improved version of these designs. Since the computation for calculating the SAD of one candidate block consists of 4 loops, the different systolic arrays proposed attenmpts to calculate two or more of these loops in parallel.

The work in [81] presents the four systolic arrays for the FSBMA algorithm where each array has a different dimension and different variable projection. The processing elements (PE) of the arrays compute subtraction, absolute computation and addition. The elements have 3 inputs (sum from previous PE, current pixel and reference pixel) and one output (sum). The output feeds to the next PE or an adder array that computes the final SAD. The arrays presented are used in conjunction with a local-memory that stores the current and search data frames and a controller that controls the array and generates the address for the memory.

In [82], two systolic arrays are presented to support two data-flow techniques: One array broadcasts the previous-block data to all the elements in the array while the current-block data is propagated. The other array broadcasts the current-block data and propagates the previous-block data. The 16 Processing Elements (PE) used consist each of a subtractor, an absolute value calculator and an accumulator. Each PE computes the SAD for one candidate vector. Registers are used to propagate data and a large comparator is used to select the best SAD of the 16 ones found at the output of each PE. Finally, a controller and an address generator are used to control the operation of the PE and to feed data into them. If a change in the block size is required, without changing the search-area, then the same array can be used as the computations carried out remain unchanged, since only the address generator requires modification. On the other hand, if the search area is changed, then multiple arrays can be cascaded to support this (allocate one area for each array)

Similarly, [79] and [80] present another set of array architectures where the k and l loops shown earlier in the code are parallelised; all absolute difference values for the SAD of one candidate block are computed concurrently and the SAD is computed using an adder tree. The previous-frame data is input sequentially, through shift registers and fed to the PEs after appropriate reordering to replace the address-generator used in the previous architectures. The shifting network of the registers is changed dynamically. Each PE has a register for storing the previous and current data and for storing interim AD. The PE has three inputs for the previous data pixels (delayed from adjacent PE, from registers, etc.) and a multiplexer to select between them. The current data is also propagated between PEs.

Architectures targeting other algorithms

Special hardware exists for running specific ME algorithms, such as the one proposed in [83] for the TTS algorithm. In this technique, 9 PEs are used each to compute the SADs of the 9

candidate MV concurrently. A column of 9 comparators is then used to select the best MV from the 9 SAD. The array described in [89] is targeted for the NTTS algorithm where 3 check-points (i.e. candidate MV) are used for the search, thus three columns of PE are used, and each column calculating the SAD of one check-point. The previous data is broadcast to every row, while the current data is propagated horizontally using programmable-delay-elements, which is required by the NTTS algorithm.

In [84] the same architecture presented in [78] is used, but a programmable address generator and control unit allow supporting alternative sub-sampling algorithms, where the pixels of the block are alternatively sub-sampled to make a *N/2xN/2* block size. Similarly in [69], the architecture from [78] is modified to enable dynamic change of the bit-width of the ME operation in order to save power. This is achieved by using different (4) clocks to the latches and flip-flops.

4.2.3. Cluster design

A flexible reconfigurable motion estimation array would support a larger number of different SAD-based motion estimation algorithms and would provide a selection of bit-width, performance, quality, power consumption and speed. This flexibility can be used at design-time as well as run-time to adapt the system to real time constrains. By examining previous hardware implementations of ME we can identify the following operations and elements in all the implementations:

- Absolute-differences (AD) calculation.
- Additions, subtractions and accumulation. Addition and accumulation are required to compute the sum-of-absolute-differences (SAD). Adders can be used alongside the AD calculators to calculate the interim SAD as in the case of the architectures given in [81]. In [81] and [82] accumulators are used to find the final SAD. Finally, in [78] [85] [69], adders are used to form an adder-tree for calculating the SAD.
- Comparison operators to select the motion-vector with minimum SAD value. The comparators can be global for the whole SAD calculator ([81]), or local for each PE module in the array ([87], [86], and [83]). The comparator should be flexible enough to support maximum/minimum calculations and general comparison (greater-than, greater-than-or-equal, equal-to).
- Registers to store the calculated AD and interim SAD values. These are useful to implement pipelined and systolic arrangement [81], [82], [71].
- In systolic implementations [82], [88] and [89], the broadcasting of data using interconnects is essential.
- Cascading of elements and modules to change the bit-width, search area and other details of the calculation.

- Multiplexing of signals to enable selecting between multiple data input signals as in the arrangements in [82].

Allowing the array to perform all the operations would allow us to implement all these different implementations. Each implementation has different characteristics in terms of throughput, area usage, bit-width and search area size, which can affect the final image quality and power consumption.

From these constrains, the following four basic elements have been designed:

- Multiplexers: 2-to-1 multiplexers with optional register at the output. Using interconnects the multiplexers can be cascaded to create larger input sizes. They also can be configured to implement a two input multiplexer, a register or a connect-through wire. Figure 4-3 (a)
- Adders: Modules supporting combinatorial 2-input additions and subtractions. An optional combinatorial absolute-difference calculators, useful for SAD based motion estimation, is also available at the output of the module. AD calculation, the difference between the two inputs can be calculated and the absolute value can be optionally selected. The output can be configured as a registered or a combinatorial circuit. Figure 4-3 (b)
- Accumulators: Sequential accumulators which can also be configured as simple combinatorial adder/subtracters. The accumulator contains an internal register. ADD, SUB, ACC, the element can be configured as adders or subtractors (combinatorial or registered) to help calculating intermediate SADs. It can also be configured as an accumulator. Figure 4-3 (c)
- Comparators: Modules enabling the comparison of two numbers producing greater-than and equal signal. Registers and logic are also available for finding and storing the minimum/maximum value useful for the minimum SAD selection. This element can compare two numbers or the input SAD with the value stored in the register, which is helpful for determining minimum and maximum values. Figure 4-3 (d)

In typical image data 8-bit values are used for representing one colour of a pixel. Hence, the adders and multiplexers are 8-bits wide and can be cascaded to produce higher bit count, in case the pixels bit-width changes. The accumulators and comparators are 16-bits wide and can also be cascaded.

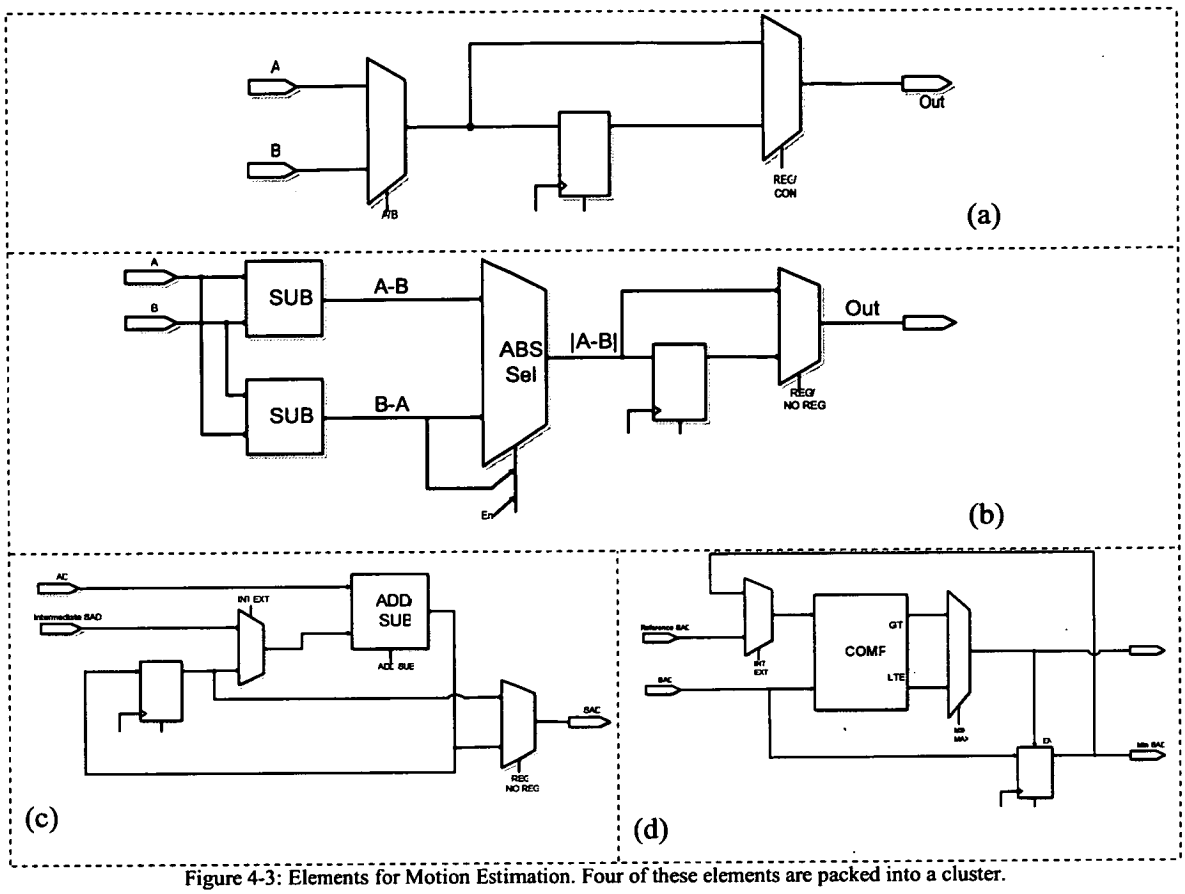


Figure 4-3: Elements for Motion Estimation. Four of these elements are packed into a cluster.

The 4 elements described above are too small to justify the overhead in interconnects needed if each element became a cluster, i.e. the area of these elements would be too small compared to the area of the additional s-boxes and c-boxes that would be built around the I/O pins (the overhead due to interconnects for typical FPGAs has been reproted to be around 90%). Hence, it was decided that 4 elements can be packed into each cluster. The main reason is that the cluster has 4 sides, and with such an arrangement all the I/O pins belonging to an element can be made avaialble on the same side. This manually created organization makes the array easier to debug, however, it might be possible to achieve better results by having a different choice of elements inside the clusters and the sides of the I/O pins. Three clusters were created as follows:

- MUX: Has 4 multiplexer elements.
- AD/ACC: Has 2 Absolute Difference and 2 Accumulator elements
- MUX/COMP: Has 2 multiplexer and 2 Compare elements

4.2.4. Cluster arrangement and interconnect mesh

The clusters were initially arranged in an array as shown in Figure 4-4 and Figure 4-5 . This arrangement follows the dataflow between the cluster from left to right, although the interconnects are bidirectional. Other array arrangements in order to provide speed and area

improvements are possible. However for the purpose of manually generated array configuration this uniform cluster arrangement was chosen. The interconnects used are based on tri-state buffers and have the full flexibility described in Chapter 2, with Fc=6 (since there are six tracks) and Fs=3. Two types of tracks are provided: Six 8-bit wide tracks for data and six 1-bit tracks for control lines. It should be noted that the multiplexers inside the clusters connecting the different elements together can be seen as a different type of interconnects. Unused elements are disabled in order to reduce power consumption. The performance of this array is measured in section 4.4.

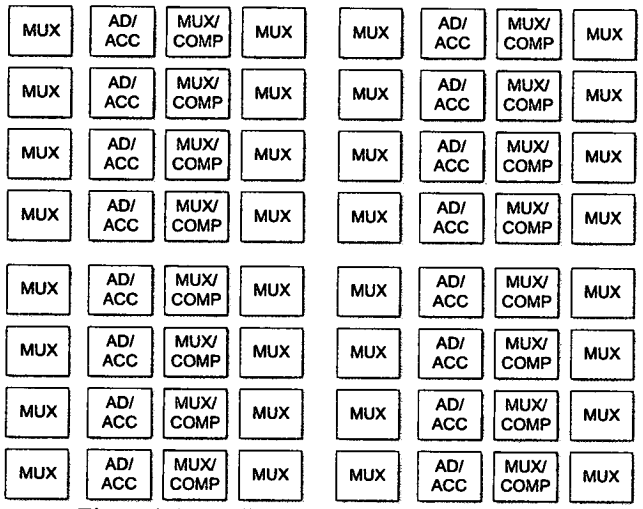


Figure 4-4: Possible array arrangement of cluster

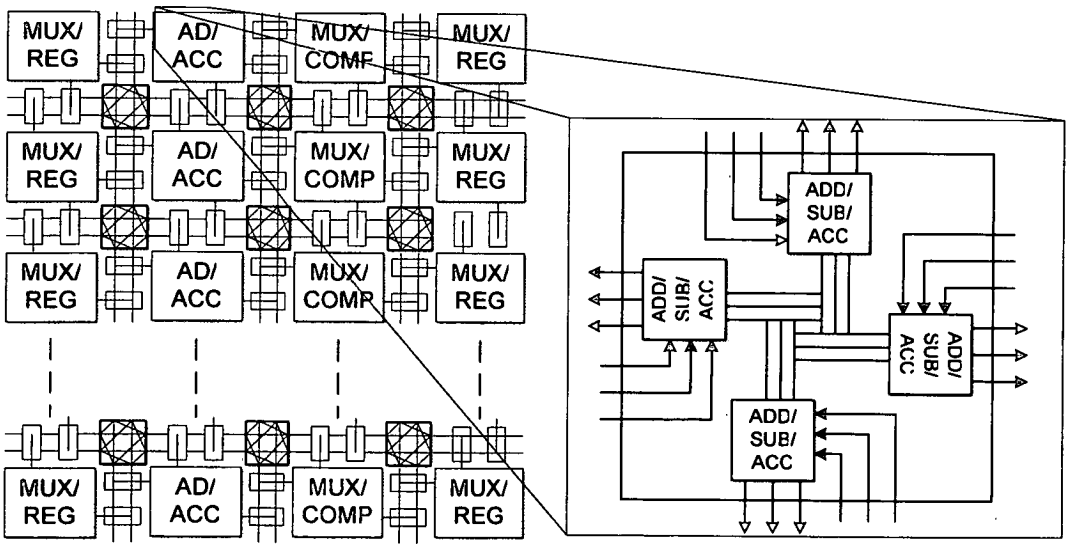


Figure 4-5: Array arrangement of cluster, with each cluster composed of 4 modules.

4.3. DSRA for DCT

4.3.1. Algorithms

Once motion estimation is calculated, the colour difference between the pixels of the two blocks is coded and transmitted. To reduce the spatial redundancy further, difference data is coded in a transform-domain. (DCT is also used to code a block that has no reference to a previous frame, in so-called *INTRA* frames). Thus, by applying a Discrete Cosine Transform (DCT) [90] to the 8x8 pixels blocks, the distribution of the data coefficients is changed in such a way that it is easier to quantise the data without losing much quality. The enerjy of the resulting DCT coefficients tends to be concentrated around the DC coefficient (at location (0,0)), and a large number of small coefficients can be effectively quantised to zero. The 2-D DCT operation is done using the following equation:

$$F_{uv} = \frac{c(m)c(n)}{4} \cdot \sum_{m=0}^{N-1}\sum_{n=0}^{N-1}\left[f_{mn} \cdot \cos\left(\frac{(2m+1)\cdot u\pi}{2N}\right) \cdot \cos\left(\frac{(2n+1)\cdot v\pi}{2N}\right)\right]$$

A N-point 1-D DCT of the input $x[]$ is defined as:

$$X_u = c(u) \cdot \sum_{i=0}^{N-1} x(i) \cdot \cos\left(\frac{(2i+1)\cdot u\pi}{2N}\right) \qquad \textbf{(4.2)}$$

Which consists of a vector by matrix multiplication. Thus for $N=8$, it can be written as:

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \\ Y_5 \\ Y_6 \\ Y_7 \end{bmatrix} = c(u) \cdot \begin{bmatrix} C_0 & C_0 & C_0 & C_0 & C_0 & C_0 & C_0 & C_0 \\ C_1 & C_3 & C_5 & C_7 & -C_7 & C_5 & -C_3 & -C_1 \\ C_2 & C_6 & -C_6 & -C_2 & -C_2 & -C_6 & C_6 & C_2 \\ C_3 & -C_7 & -C_1 & -C_5 & -C_5 & C_1 & C_7 & -C_3 \\ C_4 & -C_4 & -C_4 & C_4 & C_4 & -C_4 & -C_4 & C_4 \\ C_5 & -C_1 & C_7 & C_3 & -C_3 & -C_7 & C_1 & -C_5 \\ C_6 & -C_2 & C_2 & -C_2 & -C_2 & C_2 & -C_2 & C_6 \\ C_7 & -C_5 & C_3 & -C_1 & C_1 & -C_3 & C_5 & -C_7 \end{bmatrix} \times \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{bmatrix} \qquad \textbf{(4.3)}$$

Equation (4.3) can be seen as N parallel FIR filters with common input data X.

1-D DCT Implementation

Different popular techniques exist for implementing a 1D DCT. These techniques can also be mixed together as described below.

Dataflow Graph

A direct parallel implementation of equation (4.2) would require 64 multiplications and 56 additions (for N=8). Various schemes exist to reduce the complexity required to carry this calculation; these schemes usually reorder the input data in such a way that the computation is

simplified. This is the basis of fast DCT algorithms. These algorithms also rely on the fact that some output coefficient can be computed recursively using previously computed outputs. The dataflow presented in [91] requires 16 multiplications and 26 additions. The dataflow graph is shown in Figure 4-6, where an arrow (→) represents a subtraction and a circle corresponds to a multiplication. Similarly, the dataflow presented in [92] uses 11 multiplications and 29 additions.

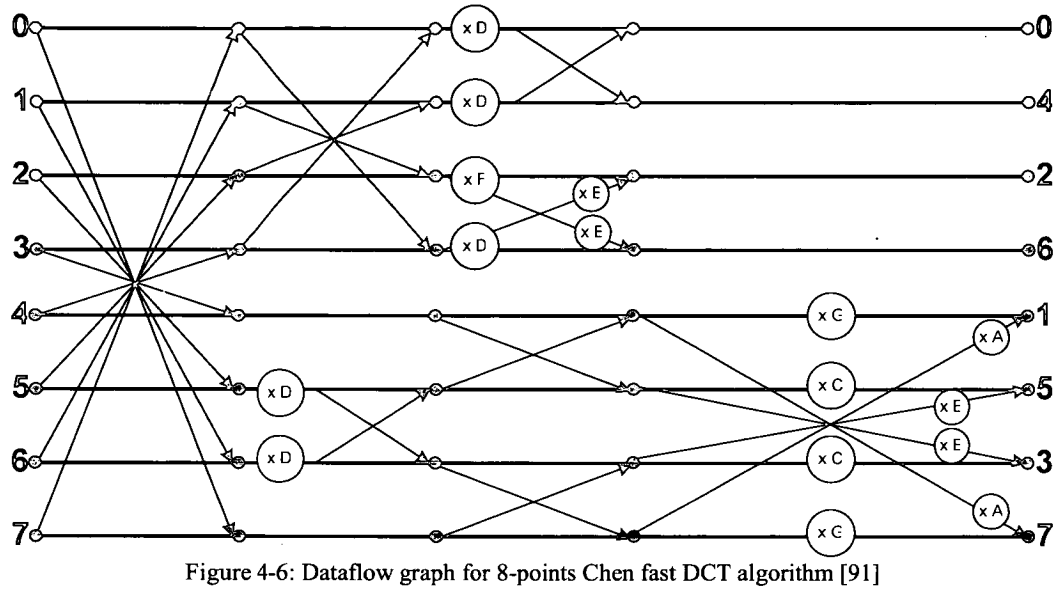


Figure 4-6: Dataflow graph for 8-points Chen fast DCT algorithm [91]

As the DCT is usually followed by quantisation (Q), it is possible to further simplify the DCT computation such that each output of the DCT is scaled by a factor. This factor is compensated for in the quantisation process and hence the name of such a DCT is as *scaled-DCT*. The work in [108] presented a flowgraph for a scaled DCT which reduces the number of multiplications to 5 and 25 additions.

Distributed arithmetic

In *Distributed Arithmetic* (DA) multiplications by fixed coefficients are carried out using a set of shift accumulates to reduce the complexity. The computation is *distributed* in the sense that the *b-th* bits from all of the input variables are processed simultaneously and not, as in conventional multiplications, where all the bits from one input variable are processed at a time. This becomes very efficient for situations where a set of input data is multiplied by several constant coefficients, as is the case in DCT and constant matrix multiplication. By using the bit representation of the input signals x_k, the following vector multiplication:

$$y = \sum_{k=1}^{K} A_k \cdot x_k$$

This equation can be reorganised and written as the following, where x_{NM} is bit M of input x_N:

$$y = \begin{array}{c} -\left[x_{10} \cdot A_1 + x_{20} \cdot A_2 + \Lambda + x_{K0} \cdot A_K\right] \cdot 2^0 \\ +\left[x_{11} \cdot A_1 + x_{21} \cdot A_2 + \Lambda + x_{K1} \cdot A_K\right] \cdot 2^{-1} \\ \mathrm{M} \\ +\left[x_{1B} \cdot A_1 + x_{2B} \cdot A_2 + \Lambda + x_{KB} \cdot A_K\right] \cdot 2^{-B} \end{array} \qquad \textbf{(4.4)}$$

As it can be seen in (4.4), the multiplication is written as bit-level AND, addition (OR) and shift operations. Each term:

$$I_b = x_{1b} \cdot A_1 + x_{2b} \cdot A_2 + \Lambda + x_{Kb} \cdot A_K$$

can be calculated using AND operators and an adder-tree. However, this is usually performed using a memory containing the pre-computed values, as shown in Figure 4-7. A fully parallel implementation of an N-point DCT using equation (4.4) would require N memory elements, each containing 2^K words. The outputs of the ROM is fed to an adder-tree with integrated shifting (done using interconnects). Several techniques and algorithms exist for reducing the amount of storage needed [93].

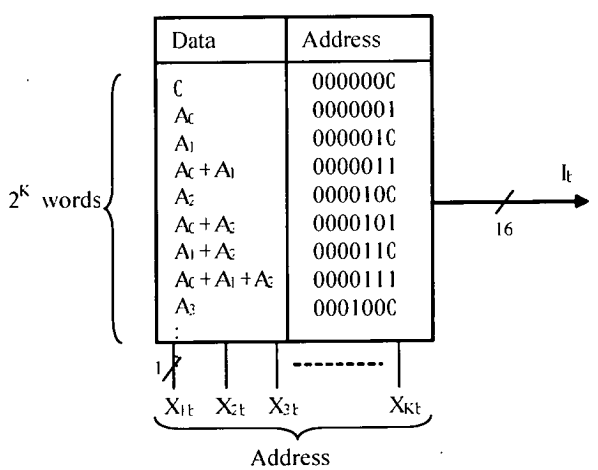


Figure 4-7: Use of memory in Distributed Arithmetic

Systolic arrays

The DCT computation can be rewritten as a recursive relationship between the DCT coefficients as described in [94]. This leads to a systolic implementation using processing elements (PEs) array, where each PE takes the result of the previous PE and applies twiddle factors multiplications and additions to get the new output. The 1-D array involves $2N$ multipliers and requires N cycles to compute a 1-D N-point DCT.

In [95], the previous recursive algorithm is merged with a fast DCT algorithm to generate an array that contains only log_2N multipliers, while maintaining the same throughput.

Digit-serial and bit serial arithmetic

To reduce the area used by bit-parallel arithmetic units, bit-serial adders and multipliers can be used. The logic used is minimised, however, B cycles are need to perform a B-bit computation. The wiring overhead and interconnects are also minimised, as only 1 or 2 wires need to be routed per interconnect. This reduces the power consumption, but prevents the exploitation of signal correlations possible in bit-parallel implementations. The Digit-serial technique is a trade-off between bit-serial and bit-parallel, where computation is carried out on several bits at a time and the required clock cycles are reduced.

Digit and bit serial arithmetic can be applied to any implementation, such as dataflow or distributed arithmetic. Bit-serial is well suited for DA, as the input data is processed at one bit from each input variable at a time (see above). When using bit-serial with DA, the adder-tree in DA becomes an accumulator.

Other techniques and combinations

Other techniques include replacing the multipliers by CORDIC calculators [96], [97], [98], which is a cost-effective method to perform rotations on vectors in the 2-D plane. This can be combined with DA as in [99].

The combination of a fast dataflow algorithm and distributed arithmetic to replace fixed-coefficient multipliers is used in [100] and [93]. This permits the implementation of a DCT with low ROM requirements.

The implementation in [101] uses 3-bit digit-serial arithmetic and DA along with a fast DCT algorithm based on the dataflow reduction. This implementation finishes the computation in 3 times fewer cycles than the bit-serial implementation, however, in terms of DA LUT memory 3 times the size is required.

2-D DCT Implementation

The computation of a 2-D DCT is generally derived from the 1-D DCT calculation. Using the row-column decomposition technique where a NxN 2D DCT calculation can be computed using two N-point 1D DCT calculations:

$$[Y_{NxN}] = [C_{NxN}] \cdot [x] \cdot [C_{NxN}]^T$$

The $C_{NxN} \cdot x[]$ calculation is done on the N rows in $x[]$, and the second DCT is done on the N columns of the intermediate result. Thus, the 2D DCT is implemented using $2N$ 1D DCT calculations and a transpose operation. Usually, two DCT modules are used. However, in some implementation only one module is implemented in order to save area, as in [102] and [101].

In other techniques only one DCT module is used to compute N 1-D DCTs, and the second DCT module is replaced by simple add and shift operations on the intermediate output result,

as in the polynomial transform technique [103], [104] and [105], the second DCT module can be replaced by simple additions and shift operation.

Alternatively, a systolic implementation can be derived by using a recursive algorithm [106]. The number of multipliers is log_2N, and no transpose memory is needed.

Amongst the implementations listed in this section, DCTs based on Distributed Arithmetic (DA) are the most promising in terms of flexibility, since DA can be adapted for other algorithms such as the Discrete Wavelet Transform (DWT); hence the DSRA designed was chosen to target DA implementations.

4.3.2. DCT using Distributed Arithmetic

A 1-D N-point DCT bit-serial DA implementation would consist of N shift-registers for parallel-to-serial conversion, N LUT memories and N shift-accumulators. All the N memories receive the same address. The 8-point 1-D DCT is shown in Figure 4-8 and Figure 4-9.

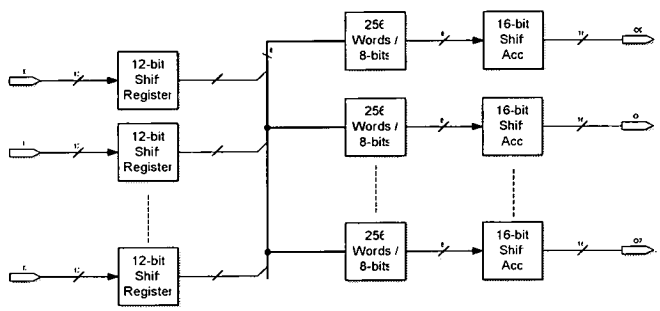


Figure 4-8: Simple DCT implementation using distributed arithmetic without memory reduction.

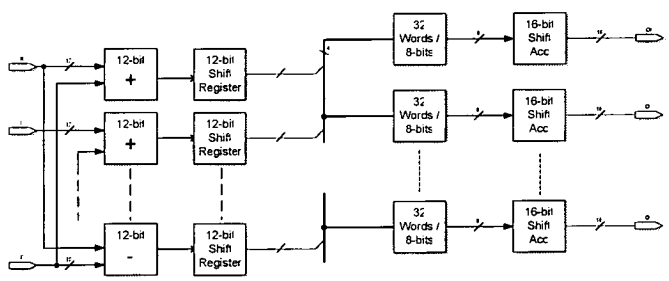


Figure 4-9: Implementation of DCT using odd-even decomposition for memory reduction.

Other DA-based implementations that the DSRA should support include a number of possible DCT implementations using DA, such as the one presented in [107] where COordinate Rotation DIgital Computer (CORDIC) computations are used to reduce the memory size, and in [101] where 3-bit digit serial arithmetic is used to improve the throughput of the array. The odd-even decomposition technique also described in [101] and shown in Figure 4-10 can be used to reduce the memory size by using adders and subtracters at the input. More details can be found in [109].

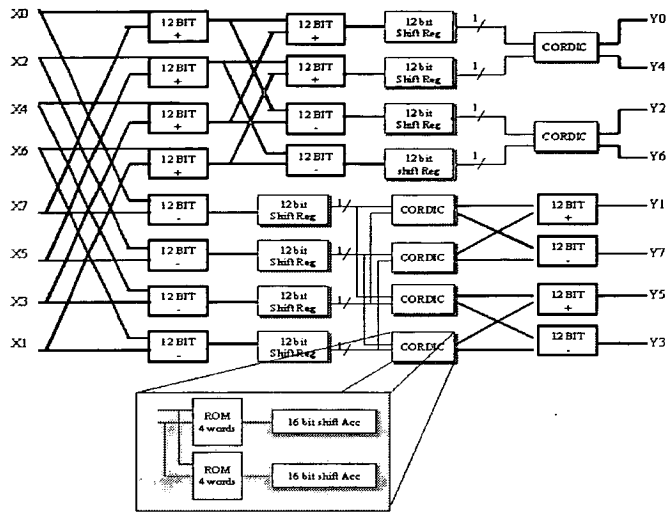


Figure 4-10: CORDIC Rotator Based 8-Point DCT Implementation mapped by Sajid Baloch to the array [109]

4.3.3. Clusters

General DA implementations require shift-registers, memory elements and shift-accumulators. Additionally, to accommodate for a wider range of algorithms such as odd-even DCT or reduced-memory DA, adders and subtracters are needed. Hence, two types of clusters have been identified and used in the proposed DSRA: Memory clusters for LUTs and add-shift clusters for making add/sub/shift and accumulation.

As described in section 9.512.5, the memory cluster is responsible for performing the pre-computed addition from Figure 4-7. The idea in DA is to make this computation pre-computed using a Look-up-table (LUT) to speed up the calculation. This is useful in ASIC designs, as the fixed LUT is translated into simple gates. However, to make this LUT programmable in the DSRA hardware, we need to use a programmable memory such as SRAM, which occupies a large area. Hence, we decided to also test the performance of a DSRA array with an adder-tree cluster that provides the same functionality as the memory-cluster by directly performing the addiction operation. In FPGAs, the LUT gets translated into a connection of fine-grain programmable gates; such a programmable logic is another potential implementation for the LUT. This was not tested, however, in theory the adder-tree solution can be seen as a more tailored (hence more efficient) version of such programmable logic that supports random fine-grain datapaths.

Memory cluster using SRAM

The memory clusters are used to implement the LUTs in the DA using SRAM. A dual-port 512-bit SRAM, organized as 64 words 8-bits per word, is used as the basic memory element. Four such memory elements are grouped together to form a 2K-bit memory cluster. The grouping is performed using logic to enable the configuration of the cluster as a memory with

all the possible geometries listed in Table 4-1. The logic used is similar to the one presented in [58]. It should be noted that each memory element can be turned off and on separately; hence, allowing the lower sized memories of Table 4-1. Each of the modules can be accessed separately, or all the 4 ones can be combined to form a big memory. In such a case, only one port needs to be used. This also reduces power consumption in unused memory.

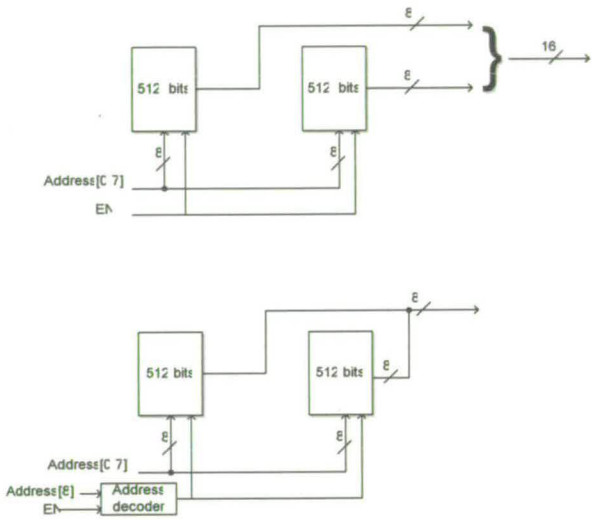


Figure 4-11: Example of combining memory-elements together vertically and horizontally.

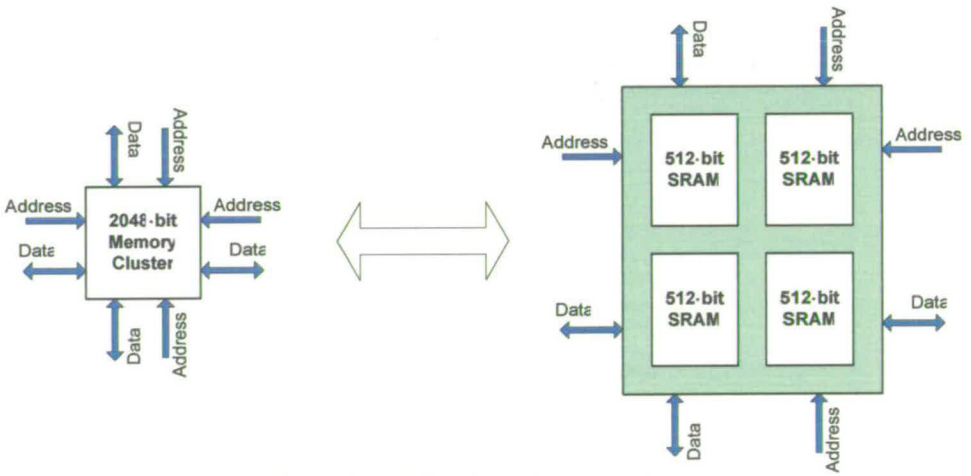


Figure 4-12: S-RAM based memory cluster

Having elements with these memory sizes enables the realization of basic DA implementations, as well as those with reduced memory described above. Clusters of memory can be further combined together using interconnects to make wider memories. Dual-port memories were chosen due to the easier configuration: Data is written during configuration on one port and read during operation on the other port. The initial content of the RAM (which reflects the coefficients) is part of the configuration data.

Table 4-1: Possible geometries achievale by reconfiguring a memory cluster.

	Bits per word			
Word Size	8-bits	16-bits	24-bits	32-bits
64	✓	✓	✓	✓
128	✓	✓		
192	✓			
256	✓			

The fact that this cluster uses SRAM makes it very flexible in terms of possible applications and not specific to DA. For comparison to the adder-tree cluster below, this cluster has an area of $0.1mm^2$ on UMC 0.18μm. Also, the SRAM from UMC can be clocked at a maximum frequency of 250MHz, which gives a response time of 4ns.

Adder-tree cluster

This cluster implements the same operation as the previous one, i.e. the computation from Figure 4-7, but using an adder-tree without precomputing the values in a table. The coefficient values A_0, $A_1 \ldots A_k$ are part of the configuration stream. As shown in Figure 4-13, each cluster contains four independent sub-modules, each summing having 8 inputs. The internal configuration to each cluster allows combining these sub-modules together. Also, in a similar way to other clusters, adder-tree clusters can be cascaded together to make bigger trees. The output can be optionally registered. Registering the output is useful in this clusters, since the output of the adder-tree has more intermediate switching activity than other clusters; the register in this case would prevent this useless activity from propagating. Also, the register would make the operation of this cluster compatible with the previous SRAM based one.

Unlike the previous SRAM-based clusters, the use of this cluster is very limited to distributed arithmetic implementations, as this is the only application that would benefit from such an arrangement. However, on UMC 0.18μm, the adder-tree cluster has an area of $47{,}258\mu m^2$, i.e. 2.13 times smaller than the SRAM based alternative. However, in terms of delays it is slower (as expected) than SRAM: If several sub-clusters are used to make an 8-input adder tree the delay was measure to be 14.02ns, which is around 3.5 times that of the above SRAM-based cluster.

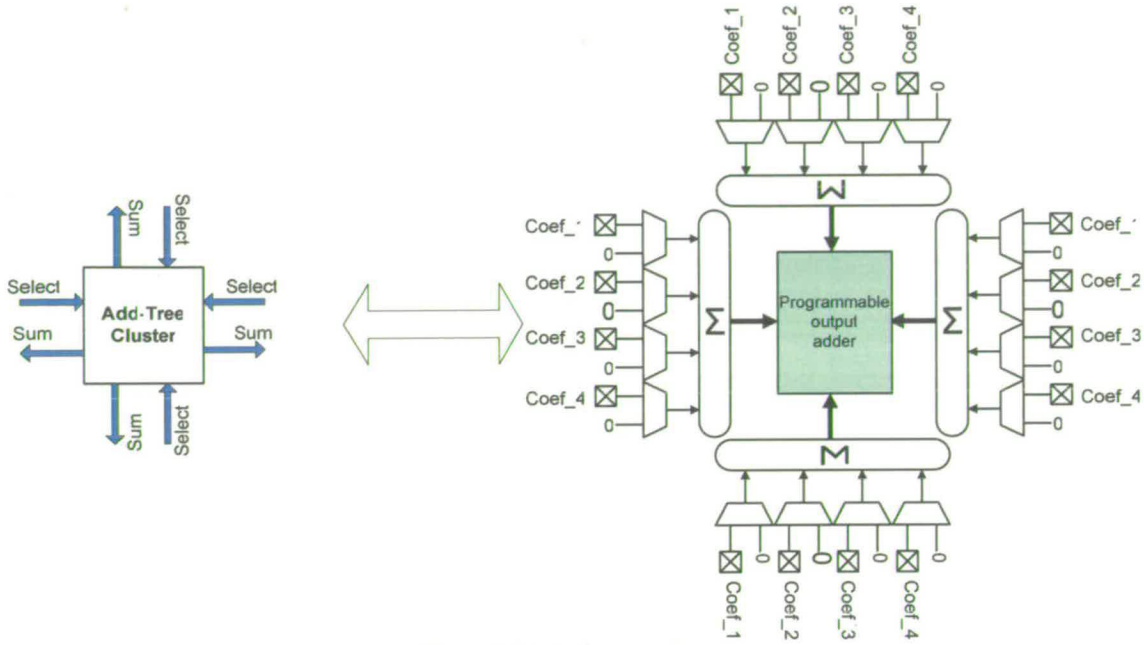


Figure 4-13: Adder-tree cluster.

Add and shift cluster

The add-shift modules provided can be configured as:

- Parallel, digit-serial or bit-serial adders/subtractors.
- Shift registers that can be used for parallel-to-serial conversion. Right and left shifts are supported.
- Accumulators with optional shift-accumulation.

Each module is 4-bit wide; four modules are grouped into a cluster and configurable switches are provided between them to support cascading to get wider bit ranges (up to 16-bits) in a similar way to the clusters used for ME. Wider operations are possible by cascading multiple clusters.

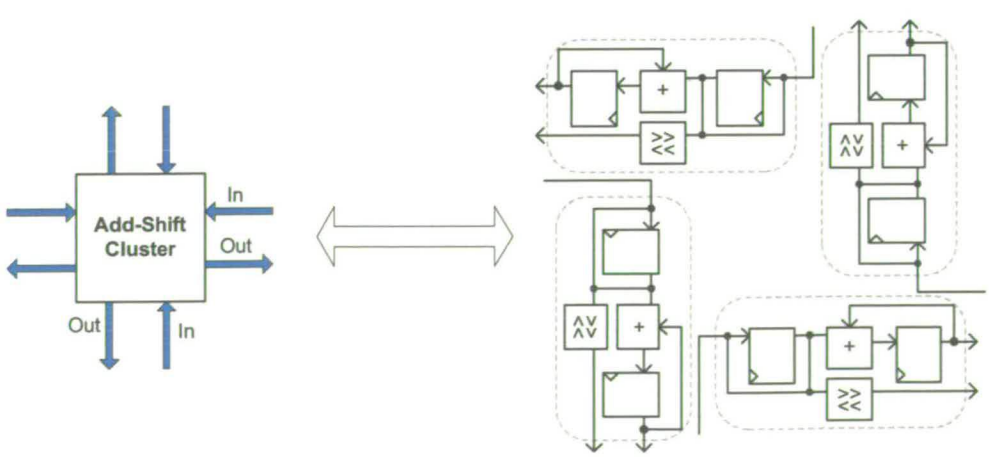


Figure 4-14: Add-Shift cluster.

4.3.4. Clusters arrangement and interconnects mesh

Again, the columns were manually arranged according to the dataflow as shown in Figure 4-15. As can be seen, the number of add-shift clusters used is three times more than that of memory clusters. This allows the mapping of a wide range of applications. The arrangement of the clusters in the array is performed at design-time and according to the required application and flexibility. The array containing the adder-tree clusters would have them in place of the memory clusters shown.

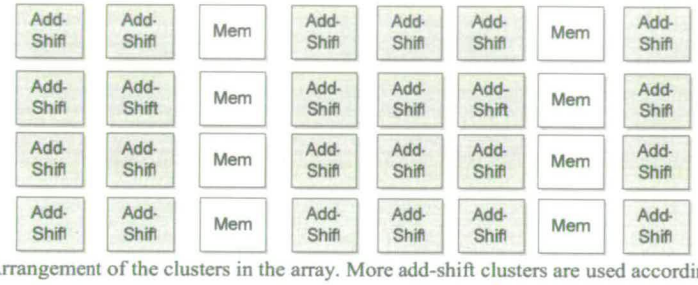


Figure 4-15: Arrangement of the clusters in the array. More add-shift clusters are used according to the needs.

The interconnects used are based on six 8-bit tracks and six 1-bit tracks provided for both data and control lines. As with the array for ME, the full flexibility interconnects from Section 4.2.4 are used, with C-boxes having Fc=6 and S-boxes having Fs=3.

4.4. Performance

4.4.1. Benchmarks

The motion-estimation architecture from [82] shown in Figure 4-16 was implemented using the module described above. In this implementation, 16 PEs are used simultaneously to compute the SAD values of 16 candidate motion-blocks. The block size is 16x16 and the search area is 32x32 pixels wide. The current motion-block data is propagated through the PEs (signal c), while two pixels from the search-area are broadcasted to the PEs (signals p and p'). Each PE is composed of a multiplexer, a register for propagation, an absolute-difference calculator, an accumulator and a comparator for selecting the minimum SAD calculated on that PE, as shown in Figure 4-16. Thus, one PE can be mapped to 3 clusters; this was manually done as follows:

- A cluster of four multiplexers and registers for implementing one multiplexer and one register.
- A cluster of two absolute-difference calculators and two accumulators for implementing one of each.
- A cluster of two comparators and registers and two multiplexers to implement one minimum-value finder.

Clearly, the mapping of elements is not the most efficient in terms of area usage since it was performed manually. An intelligent automatic mapping process, similar to the ones found in current FPGA implementation software would have produced better results in terms of area and timing.

To implement a full ME hardware, further clusters for implementing the generic control functions such as counters and state machines are needed for the purpose of this benchmark; these controller has been simulated as hardware. The ultimate goal of the project is to provide a library of clusters that include elements for executing Finite State Machines (FSMs) as described in the derived project [110].

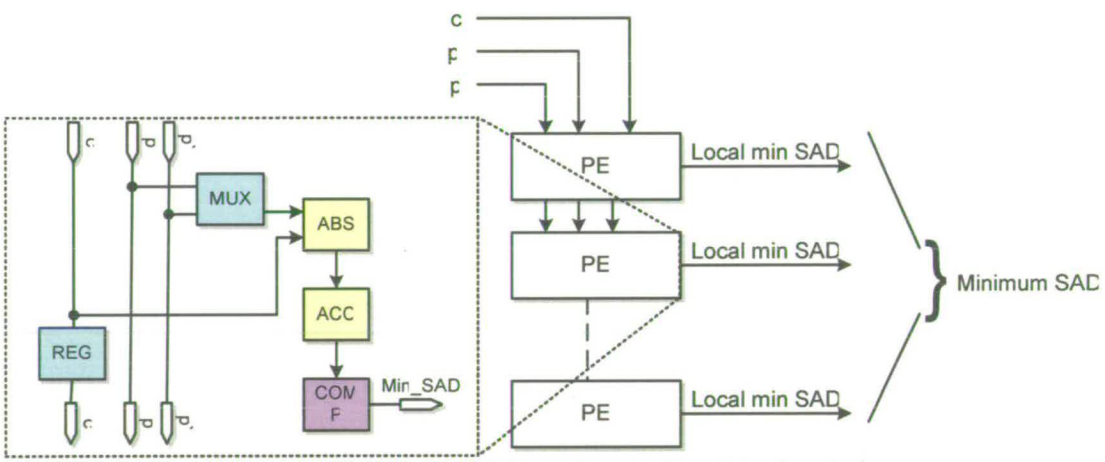


Figure 4-16: Mapping of a PE from [82] using 7 modules from 3 clusters.

The simple 8-point 1-D DCT calculation without memory compression and the DCT with odd-even decomposition described in Figure 4-8 and Figure 4-9 were implemented on the RA. The DCT is implemented using 12-bits input coefficients and 8-bits output coefficients from the LUT, which results in a 16-bit output values. The first DCT without memory compression has been manually mapped such that:

- A 12-bits shift register is mapped to three add-and-shift elements part of one cluster.
- A 2-Kbit memory is mapped to four memory elements found in one cluster.
- A 16-bit shift accumulator is mapped to four add-shift modules part of one cluster.

In the second DCT implementation with odd-even decomposition the mapping was similar to the previous one but with the following differences:

- The 8-bit adder/subtractor at the input is mapped to two add-and-shift elements part of one cluster.
- The 32x8 bit memory is mapped to one 256x8 bits memory element found in one cluster.

Both implementations were carried out using the Memory-LUT and Adder-tree version of the array. Other DCTs and DWTs were implemented by Sajid Baloch on the same array as part of his work [109]. However, the performance of these implementations were not measured and not listed here.

The same benchmarks were also implemented using standard hardwired ASIC and using a commercial Xilinx Virtex-E FPGA. ASIC and Virtex-E: All of these systems use a 0.18μm CMOS technology and are powered at 1.8V. In the case of the DCT, they all run at 10MHz, and for the ME, the operating frequency is 30MHz. The power, area and timing measurements for the hardwired and the DSRAs implementations are done using post-layout simulations vectors with typical switching activity and accurate parasitic and load information. Synthesis was performed with *Synopsys DesignCompiler*, the layout with *Cadence Silicon Ensemble*, power estimation with *Synopsys PrimePower* and timing evaluation with *Synopsys PrimeTime*.

The area estimation on the Xilinx Virtex-E FPGA is based on the estimate that the area of one slice, its surrounding routings (C-boxes and S-boxes) and its belonging configuration memory occupies 3303 μm^2. This estimation was found by taking the approximate area of the Virtex-E core without I/O pads, memory blocks and clock buffers (from a die photo[111]) and dividing it by the total number of slices in the chip. The power measurement of the FPGA's logic was made using *Xilinx XPower*. The power includes only the logic cell and its belonging configuration memory, but not any I/O port, clocking buffers or other memory elements.

The performance in terms of area, power consumption and maximum frequency is shown in Table 4-2 for the ME implementation and in Table 4-3 and Table 4-4 for the DCTs. In the case of the DCTs, the values are measured for one row only of the array; the result for a full 1D DCT or a 2D DCT would be similar.

Table 4-2: Performance of the implementations of one ME processing-element from [82]

	.18μm ASIC	*DSRA*	Xilinx's Virtex-E
Area (μm^2)	8,594	32,207	178,362
Power consumption (mW)	0.68	1.08	4.37
Max Freq. (MHz)	440	111	90

Table 4-3: Performance of the simple DCT implementation on DA array with SRAM

	.18μm ASIC	*DSRA & SRAM*	*DSRA & Adder-tree*	Xilinx's Virtex-E
Area (μm^2)	17,483	212,135	172,212	234,510
Power consumption. (mW)	0.52	1.922	1.531	3.2
Max Frequency (MHz)	210	77	68	50

Table 4-4: Performance of the odd-even DCT implementation on DA array with SRAM and array with Adder-Tree

	.18μm ASIC	*DSRA & SRAM*	*DSRA & Adder-tree*	Xilinx's Virtex-E
Area (μm^2)	10,518	235,234	143,872	267,725
Power consumption. (mW)	0.48	1.50	1.28	2.9
Max Frequency (MHz)	250	77	68	66

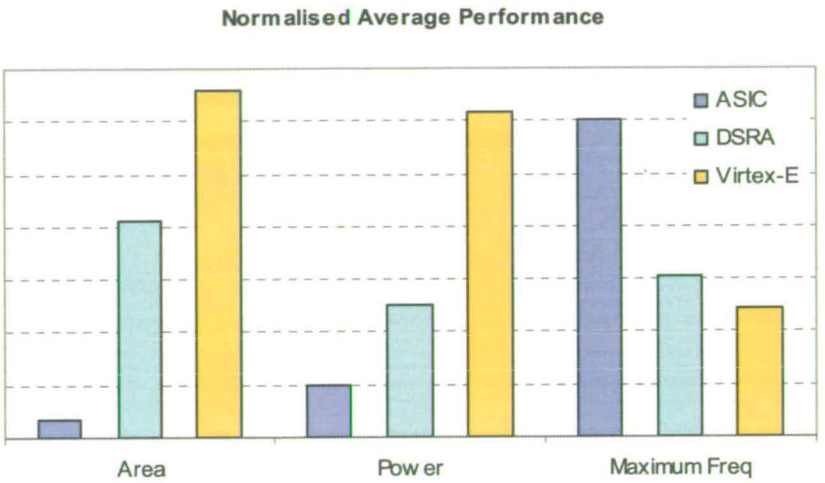


Figure 4-17: Average performance of DSRA in all benchmarks

Area

From Figure 4-18 below, it can be seen that the relative area of the DSRA compared to ASICs and FPGAs greatly depends on the application running and design of the clusters in the DSRA. On average (see Figure 4-17) the area of the DSRA is 12 times that of the ASIC, while being around 60% of the FPGA's occupied area. The relative performance figures are better in the case of the motion-estimation implementation, as they are closer to the ASICs one than the FPGA (the DSRA is only 3.7 times larger than the ASIC).

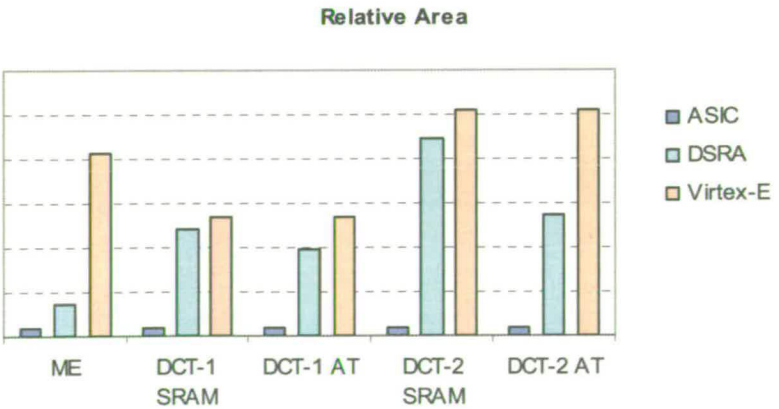


Figure 4-18: Relative area comparison of DSRA wit ASIC and FPGAs.

Power consumption

When examining the power consumption we can see that the power consumed by the DSRA is indeed a middle-ground between ASICs and FPGA: It is on average 3 times lower than FPGAs while 2.5 times larger than ASICs. Again, this also depends on the DSRA and implementation – in the case of DCTs with SRAM-based clusters, the power consumption is only 40% less than in FPGA; this is caused by the fact that using SRAMs for implementing such tables is not much more efficient than using the LUTs in the FPGA.

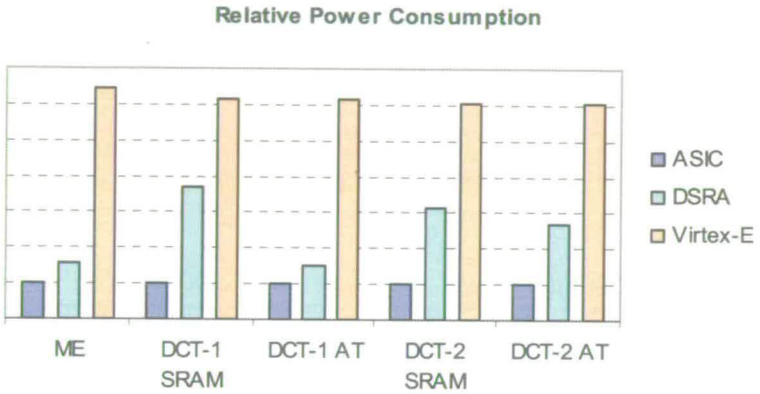


Figure 4-19: Relative power comparison of DSRA wit ASIC and FPGAs.

Timing

From a timing perspective, the implemented DSRAs are on average 20% faster than the FPGA, while being 3 times slower than ASICs. The best speed is observer for the DCT with SRAM case where the DSRA achieve around 40% the speed of ASIC. This increase in delays comes as a price for the increased flexibility due to the extra over head introduced in the reconfigurable switches and the higher-loads and longer routings.

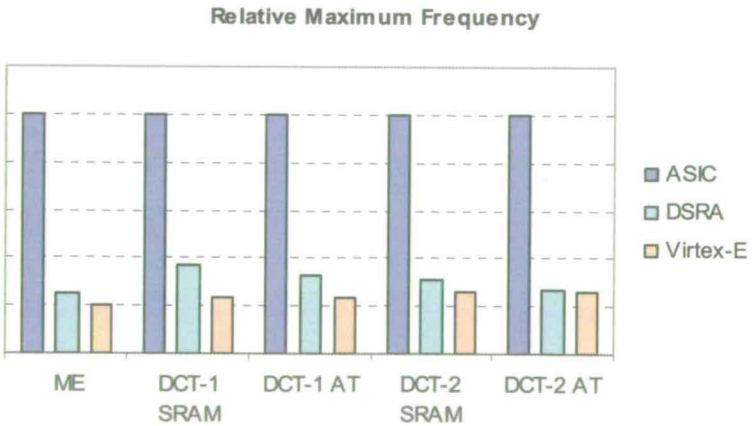


Figure 4-20: Relative maximum frequency comparison of DSRA wit ASIC and FPGAs.

4.4.2. Comparison of the DCT implementations

When comparing the DCT with adder-tree cluster and the DCT with SRAM-based clusters, it can be clearly seen that the SRAM achieves slightly higher speeds (13% higher) at the cost of much higher area (increases between 25% and 60%) and higher power consumption (20% higher). The higher area in the case of the SRAM is not only caused by the large space occupied by memories, but also due to the fact that the size and dimensions of the SRAM cluster are larger than the add-shift clusters. Hence, organising them uniformly into an array leads to wasted area. This is not the case for the adder-tree cells, as they have a similar area to the add-shift clusters.

The odd-even decomposition in the DCT requires less memory due to the smaller LUTs; however, an extra adder/subtractor is required per row. This is reflected in the area used by the second implementation, which is 10% higher than the first one.

Power consumption is reduced by 22% in the second implementation due to the fact that the adder/subtractor consumes less power than the large memory. The maximum frequency is the same in both implementations, due to the fact that the largest delay is between the output of the shift-registers and the output of the shift-accumulator, and not at the input. It is also possible to implement the adder/subtractor as bit-serial elements after the shift-register, but this may introduce extra delay.

Similar results are found when comparing the ASIC and the Virtex-E implementations of both DCTs.

4.4.3. Measurement of overhead

When compared to hardwired solutions, the added programmability comes at the expense of an overhead in power and area consumption. In this case this overhead can be effectively seen as the average contribution of the interconnects (C-Boxes and S-Boxes) and the configuration bits is to the total area and power of the array.

Power overhead

When modules and clusters are unconfigured and if there is no activity at their inputs, they exhibit only static power consumption. In the case of unconfigured C-boxes, some switching power is dissipated when the output of the cluster connected to the C-box is switching.

The total static power consumption of the array was measured to be only 0.03% of the total power consumption. Hence we can consider that static power consumption of unconfigured fabric to be negligible when compared to the total power consumption. This assumption is only valid for 0.13um technology and above, as smaller technologies would have a larger value of leakage power.

Figure 4-21 shows the total power consumption of one add-shift cluster and its associated C-Box and S-Box. The values shown are the average of both the shift-register and shift-accumulator used in one row of DCT. Highly similar values are found when examining other clusters in the DCT or ME array, except the Memory clusters in the DA array, since SRAM consumes a high energy compared to logic.

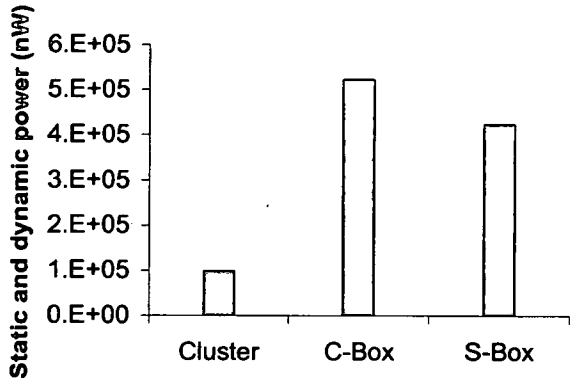


Figure 4-21: Distribution of the average power consumption between an add-shift cluster and its associated C-box and S-Box.

From the graph it can also be concluded that the power consumed by the cluster is only 9% of the total power, while the C-Box consumes 50% and the S-Box 41%. This is expected due to the high number of switches and buffers introduced in the signals and due to the long routing. This could be improved by reducing the flexibility of the boxes taking into consideration that the flexibility is not decreased greatly [43]. Hence, the next step in future power reductions would be in optimizing the interconnects.

Area overhead

Similarly, Figure 4-22 shows the area overhead used to make the hardware reconfigurable. The add-shift cluster occupies only 6% of the total area while the C- and S-boxes occupy 50% and 44% respectively. As it can be seen from the graph these area values include the area occupied by the configuration registers, which represents a large percentage of the area of the boxes. The total area can be reduced considerably if the flexibility of the C- and S-boxes is lowered: this would reduce the size of the configuration memory as well as area switches.

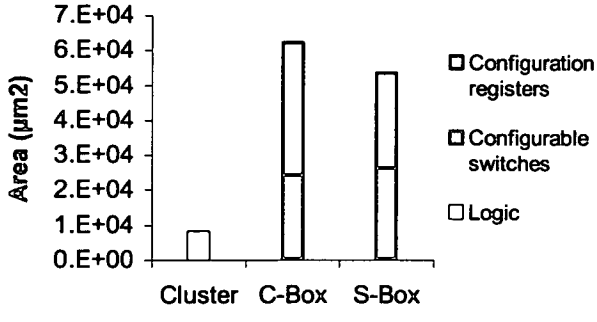


Figure 4-22: Area of add-shift cluster and its associated C- and S-boxes.

Using a data coding style to compress the bit-stream in the configuration registers, e.g. usage of a decoder in the C-Boxes to allow connecting a pin to one track only would reduce substantially the number of configuration registers required, while maintaining the same number of configurable switches. This would reduce the area at the expense of removing the option of connecting a pin to multiple tracks.

4.5. Conclusion

In this chapter, two DSRAs for multimedia application were designed and several benchmark-circuits mapped to them. The first array targets the *Motion Estimation* computation, while the second is for the *Discrete Cosine Transform* and Distributed Arithmetic applications. Initial results showed that the proposed technique of building-up reconfigurable arrays by creating application-specific clusters and combining them with an interconnects mesh provides a good compromise between hardwired and FPGA solutions: The DSRA was assessed to provide on average 3 times less power, 60% less area and 20% less delays than FPGAs, while having consecutively 2.5, 12 and 3 times more power, area and delays than ASIC. The flexibility provided by the array is limited between the boundaries of the application it was designed for, which makes its flexibility somewhere between FPGAs and ASICs.

Table 4-5: Advantages and disadvantage of the DSRA to FPGA, ASIC, and DSP

DSRA vs. FPGA
▪ Lower area ▪ Much lower power consumption ▪ Higher frequency ▪ Less flexibility
DSRA vs. ASIC
▪ Much higher flexibility ▪ Higher power consumption ▪ Higher area ▪ More delays
DSRA vs. DSP
▪ Better performance ▪ More difficult to program, integrate and debug than processors

However, DSRAs have several limits which could curb their chance of becoming the ultimate architecture for future mobile devices. The most important limitation is the way the implementations are designed, i.e. through a HDL netlist; to implement an algorithm the designer is required to have knowledge in hardware design. Since it takes a long time to design on a hardware level, a better solution for future architectures would be to provide a solution that can be easily programmed through a high-level language such as C/C++.

On another level, the way the configuration memory was implemented as a shift-register makes the whole reconfiguration process time-consuming and limits the dynamic reconfiguration ability of the array. This is due to the high number of configuration bits required. Finally, as measured, the reconfigurable interconnects consume around 90% of the total power and area of the array. This high overhead in flexibility is acceptable in FPGAs, but it should be lower on domain-specific architectures. Some of these limitations are addressed in the following chapters.

Chapter 5:

Synthesisable interconnect customisation for DSRAs

As seen in the previous chapter, further performance improvements in the DSRA's interconnect and configuration memory need to be investigated in order to allow further reductions in area and power consumption. Such performance improvements can be achieved by making the interconnect and its configuration memory more tailored to the application, in a similar way the clusters were designed.

In the previous chapter it was measured that the island-style non-segmented programmable interconnects used occupied up to 91% of the total array area and power consumption. Such high ratios are usual for generic fine-grain FPGAs, however this is too high for the purpose of embedded coarse-grain arrays. The C-Boxes and S-Boxes making the interconnects share the total area and power between them by around 50% and 41%, respectively.

The main inefficiency occurs when trying to build synthesisable interconnects and configuration memories having the same functionality as the ones found in typical FPGAs.

The use of standard-cells libraries limits the possible circuit designs of the programmable switches, since the pass-transistors used in typical FPGAs [46] have to be replaced by synthesizable cells such as tri-state buffers or multiplexers. This significantly increases the area, power consumption and delays: two tri-state buffers forming a bidirectional switch have nearly 8 times the area of a single pass-transistor. This is similar to synthesisable memory; synthesisable alternative for SRAM-cells such as flip-flops or latches can occupy up to 2.7 times more area. As described in [59], a possible solution is to augment the standard-cell library with handcrafted FPGA-friendly cells. However, this reduces the portability of the array between different fabrication technologies.

The approach in this chapter is to change the design of interconnects so that they become customised to the application in order to reduce the area and power requirements. To verify the validity and performance gained by such a strategy, the DSRA created for the DCT computation is taken as an example.

5.1. Proposed designs

S-Boxes designed using pass-transistors take advantage of the fact that that pass-transistors act as bidirectional programmable switches. To design such a synthesizable bidirectional switch (see Figure 3-3 and Figure 3-8), two tri-state buffers are needed. A single tri-state buffer is a uni-directional switch. A similar uni-directional switch can be implemented using multiplexers.

In this work, only the design of the *6W switch-point* [42] from which the switch-box is made up is investigated. The *6W switch-points* are connected together using the standard *Subset* s-box topology shown in Figure 5-1, as this was initially measured to provide better routability results than other topologies such as the *Universal* and *Wilton* ones [51] [53]. The boxes with full directions have a flexibility of Fs=3. This value was initially chosen for simplicity and for creating interconnects that have the same functionality as the ones found in standard FPGAs. (It should be noted that this flexibility measure does not apply to the s-boxes with reduced directions explained below, as these would have different values for each side.)

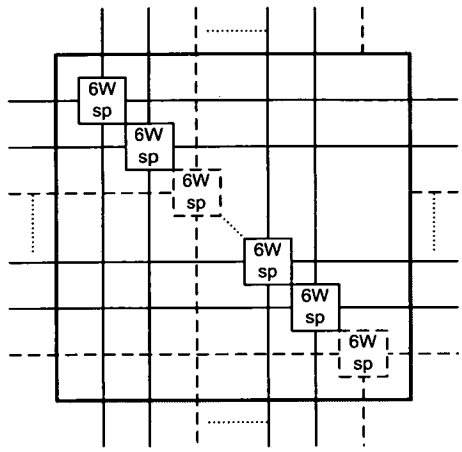


Figure 5-1: S-Box formed out of 6W switch-points arranged in a subset topology.

The following 7 variations of s-boxes designs are compared together. They use both tri-state buffers and multiplexers inside their switch-points:

(1) *All directions, tri-state*
(2) *All directions, multiplexers*
(3) *All directions, tri-state with reduced cfg memory*
(4) *Reduced directions, tri-state*
(5) *Reduced directions, tri-state with reduced cfg memory*
(6) *Reduced directions, multiplexers*
(7) *Reduced directions, multiplexers and tri-state*

The performance of these designs is compared later in section 5.2.

5.1.1. Full directions using tri-states

As was shown in Figure 5-2, this design attempts to create bi-directional switches that connect any two sides together by using tri-state buffers. The switch-points shown have the same functionality as the basic switch made using pass-transistors in generic FPGAs; hence this switch has the relatively highest flexibility when compared to the rest of the proposed below.

One switch point requires **12 configuration bits**.

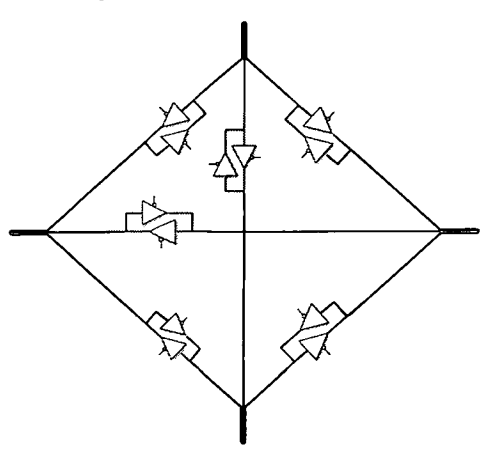


Figure 5-2: 6W switch-point using bidirectional tri-state buffers. 8 configuration bits

5.1.2. Full directions using multiplexers

This switch has the same functionality and flexibility as the previous one but uses a *3-to-1* multiplexer and one tri-state buffer per port to implement this. A similar design was presented in [112]. The tri-state buffers at the outputs are still needed since the track is driven by multiple sources.

One switch point requires **8 configuration bits.**

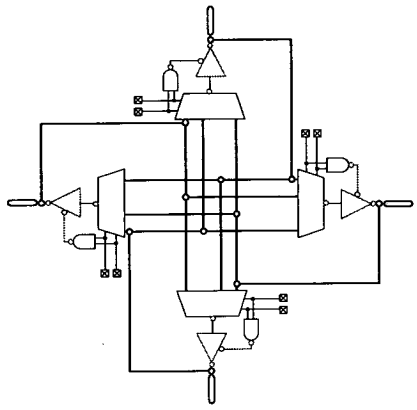


Figure 5-3: 6W switch-point with full directions using multiplexers

5.1.3. Full directions using tri-states and compressed configuration memory

Since the area cost per configuration memory bit is high, area optimizations might be achieved by compressing the memory content: e.g. the number of configuration bits needed in switch (1) can be reduced by compressing the redundant states, since only 2 bits are required per side to select which of the 3 other sides, if any, has to be routed through. Hence, decoders are used in here to reduce the number of configuration bits from 12 to **8 configuration bits.**

5.1.4. Reduced directions using tri-states

Depending on the placement of the components on the array the data flow can be more intense in some directions than others. This is especially true when routing for our case of coarse-grain circuits where the direction of the data-flow is predictable, unlike the case of random logic circuits in FPGAs. Hence, switches (4)-(7) favor some directions over others. It should be noted that switch-points with reduced directions are still able to perform all the possible connections between two sides by using two tri-state buffers in a row, but this requires more resources and creates more switching activity in the wires, as measured in section 5.2.

As shown below in Figure 5-4 for this switch, two types of switch-point are proposed, each allowing connections only in specific directions. The two types of switch-point are both used

in different ratios inside the switch-box as shown in Figure 5-5, which allows the creation an overall switch-box that accepts more connections from left-to-right and top-to-bottom.
One switch point requires **6 configuration bits**.

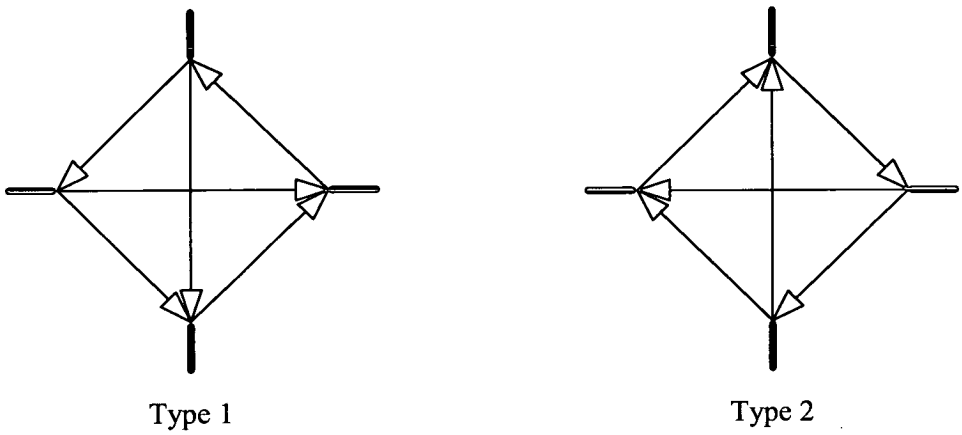


Figure 5-4: Two possible arrangements for the 6W box using tri-states

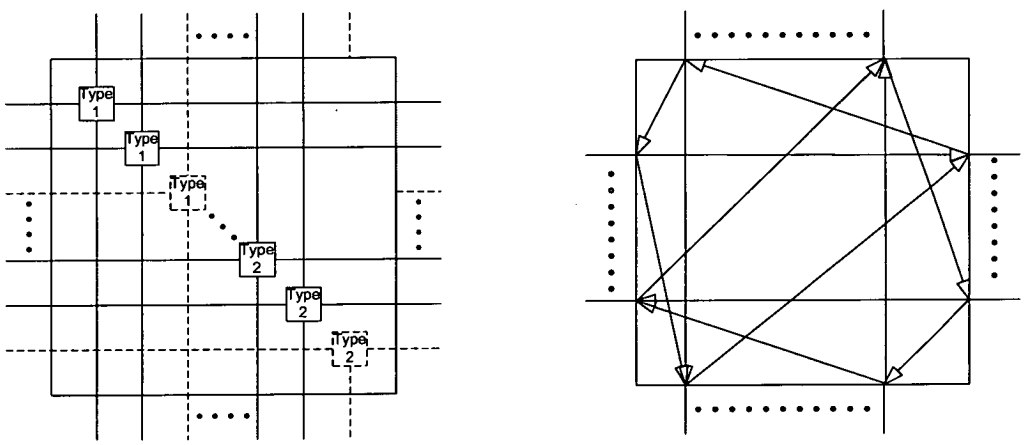


Figure 5-5: Possible arrangements using the two types of 6W boxes

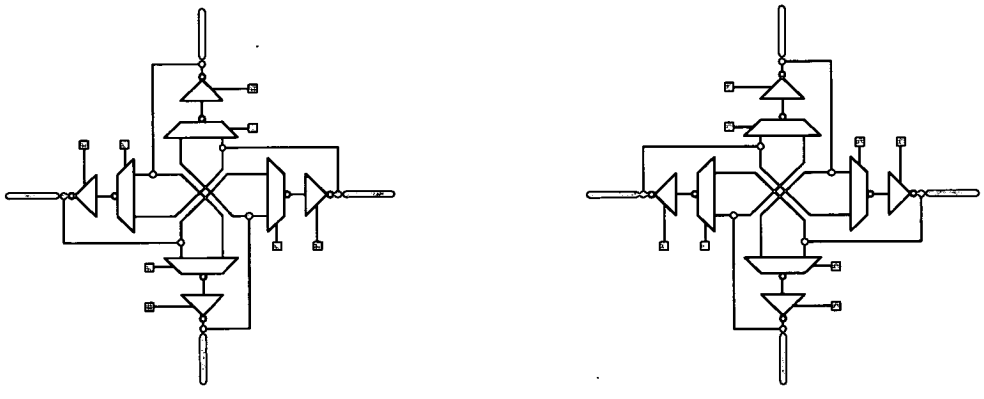
5.1.5. Reduced directions using tri-states with compression

In a similar way to switch (3), this switch reduces the configuration bits required in switch (4) from 8 down to **4 configuration bits**. However, the flexibility is reduced as only two tri-state buffers are allowed to be on at the same time, which also decreases the routability of the design.

5.1.6. Reduced direction using 2-to-1 multiplexers

As seen below in Figure 5-6, the use of 2-to-1 multiplexers allows the switch to have a larger flexibility than the buffer-based switch (4). Each multiplexer is followed by a tri-state to allow disabling the connection.

One switch point requires **8 configuration bits**.

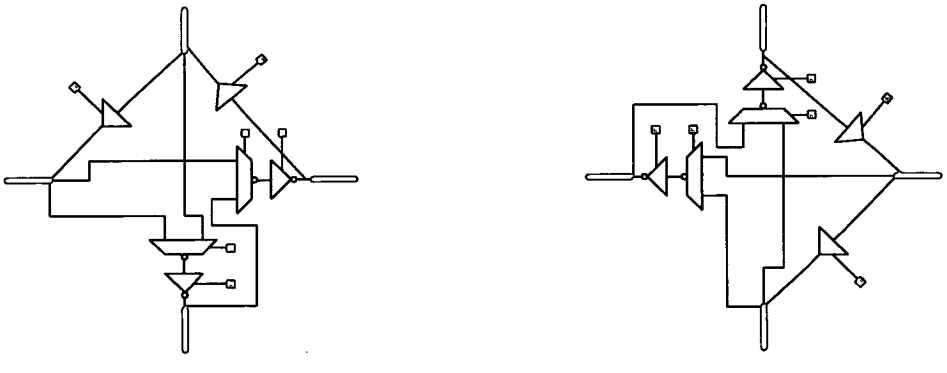


Type 1 Type 2

Figure 5-6: Two possible arrangements for the 6W switch-point using 2-to-1 multiplexers

5.1.7. Reduced directions using both tri-states and 2-to-1 muxes

This switch uses both multiplexers and tri-state buffers to create a switch with the same functionality as (4), as shown in Figure 5-7. One switch point requires **6 configuration bits**.



Type 1 Type 2

Figure 5-7: Directional 6W switch-points using both tri-states and multiplexers.

5.2. Performance evaluation

In order to identify the most suitable 6W switch-point design, the performance of each circuit is measured in terms of area, power, delays and routings overhead. An array with each type of switch-box was generated and a sample circuit was mapped on it. The benchmark circuit used is the DCT implementations mapped to the DSRA designed for Distributed Arithmetic (Section 4.3). The test conditions are slightly different from the ones in the earlier in chapter: A UMC 0.13μm technology is used as opposed to UMC 0.18μm. The 0.13μm technology has a higher leakage power consumption which should provide an evaluation better suited to future technologies with high leakage power.

5.2.1. Area

The area of the switch-boxes can be split in two parts: The area needed for the actual switches and the area required by the configuration memory. The total area of these switch-boxes and the contribution of the switches and configuration memory are shown in Figure 5-8. The values shown are for a switch box containing 12 1-bit tracks and 12 word-wide tracks. The configuration memory used is based on flip-flops; other alternatives such as latches would require slightly less area as described in Section 3.5. The area measurements also include the overhead in the metal routing required, which varies due to changes in the number of wires inside each box.

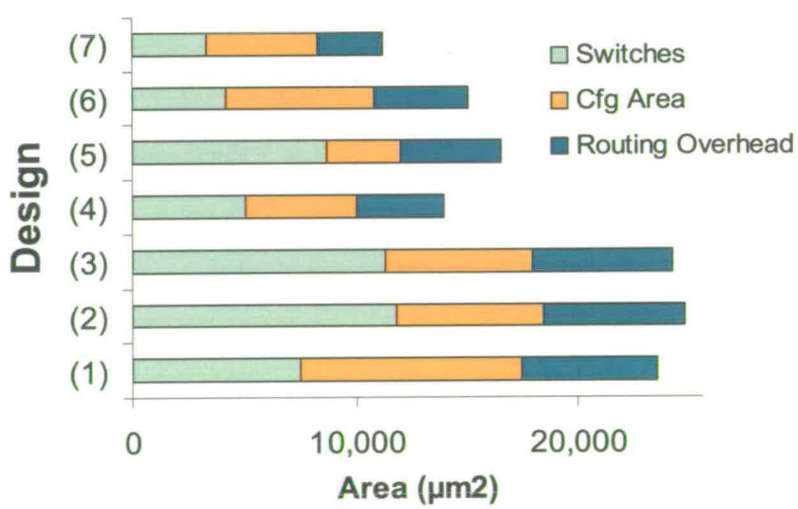


Figure 5-8: Area of Switch Boxes with contributions of switches, configuration memory and metal routing.

As expected the highest areas are consumed by the switch-boxes having full directions (1), (2) and (3). Implementation (2) with the 3-to-1 multiplexers has the highest area, which is 5.2% more than that of (1). Implementation (3) shows that no gain is achieved by compressing the configuration memory, as the area in (3) is 2.8% higher than in (1), due to the area occupied by the decoding circuit which is higher than what would have been taken by configuration

memory. These results depend on the number of bits in the word track of the array as explained at the end of this section. The result also depends on the design library and cell-geometries used: other libraries used (UMC 0.18μm) showed results where (2) had up to 11% lower area than (1) for the same widths of tracks.

The switch-boxes with reduced directions have considerably less area than the full directions ones. Implementation (4) has half the area used by (1) since the number of switches and configuration bits is halved. In (5), for the chosen number of tracks, the area savings in configuration memory is less than the area occupied by the decoding circuit used, and hence (5) is 18% larger than (4). The use of 2-to-1 multiplexers in (6) reduces the area taken by switches when compared to tri-state buffers in (4); however, more configuration bits are needed which make the overall area of (6) 8% higher than (4). Finally, implementation (7) has the lowest area, which is 20% smaller than (4), since the switches area is reduced by using 2-to-1 multiplexers and the number of configuration bits is kept the same.

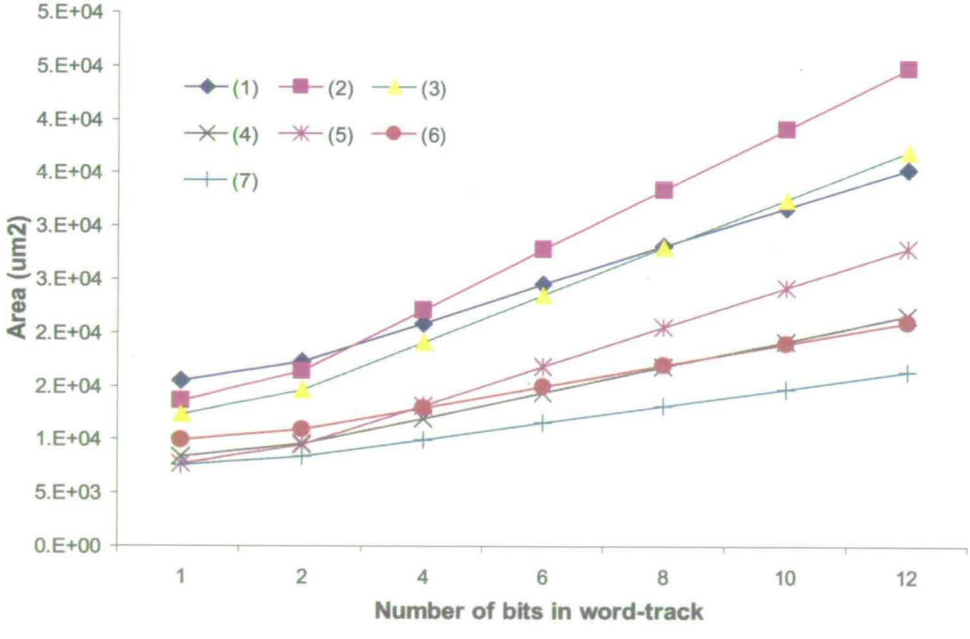


Figure 5-9: The routed area vs. number of bit in the word tracks.

The graph in Figure 5-9 shows the relationship between the area of the boxes and the number of bits in the word-tracks. It should be noted that when the bit-width of the word track is increased, the number of configuration bits remains constant and only the area occupied by the switches is increased. It can be seen that the use of compressed configuration memory as in (3) and (5) only offers area advantages for bit-widths below 8 and 4 respectively. The implementations with reduced directions have always a lower area; switch-box (7) has the smallest area for all bit-widths of the word-track.

5.2.2. Power consumption

The total power consumption measured for each type of switch-box is shown in Figure 5-10. It can be clearly seen that the introduction of the multiplexers in implementation (2) increases the total power consumed by 29%. Similarly, implementation (3) has a slight increase of 3% in power due to the presence of the decoders, even though the decoders are not in the data path and hence do not get as much switching. This increase is due higher leacker power cause by the larger area. The same slight increase can be observed between (4) and (5).

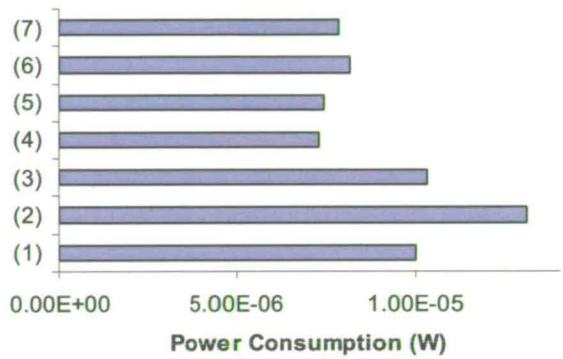


Figure 5-10: The typical power consumption per switch-box type

The power consumption in (4) is reduced by 27% when compared to the one in (1) since the load on the input lines has been reduced. It should be also noted that when using the switches with reduced directions extra routing is required on the array, and hence more power is dissipated in other switches-boxes on the array (the values measured is the average of all the switch-boxes). Switch-boxes (6) and (7) consume 8% to 12% more power than (4), while having around 20% less power than (1).

5.2.3. Delays

The delays in implementation (1) are the lowest as the switch has a high flexibility which generates short routed interconnects (see Figure 5-11). Switches (1) and (3) have both the same delays since the decoding circuit in (3) does not affect the data path signals. The use of 3-to-1 multiplexers in the data path in switch (2) increases the delays considerably by 37% when compared to tri-state buffers. The switch-boxes with reduced directions only show between 7% and 14% more delays than the full switch box due to the longer routings created. Furthermore, the use of 2-to-1 multiplexers in (6) does not add as much delay as the 3-to-1 multiplexers in (2).

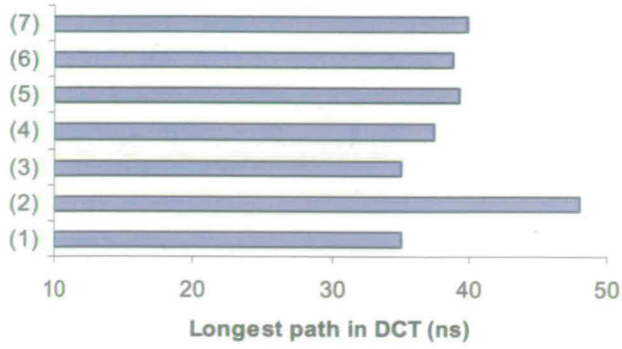


Figure 5-11: The longest path in the DCT implementations using each switch-box type.

5.2.4. Routability

The ratio of Type 1 and Type 2 blocks in switch-boxes with reduced directions (implementations (4), (5), (6) and (7)) has an effect on the routability of the design depending on the data-flow. Changing this ratio has an effect on the total wirelength of the routed design, as measured and shown in Figure 5-12 for switches (4) and (7) and in Figure 5-13 for switch (6). It can be seen that for implementations (4) and (7) the lowest wirelength is achievable when around 65% of the switch blocks are of Type 1. For switch (6) the minimum wirelength occurs when around 60% of the blocks are of Type 2.

The routability of each switch-box type is shown in Figure 5-14. Implementations (4), (5) and (7) with optimized ratios have a wirelength around 12% higher than the implementations with full-directions. Using switch (6) with the optimized ratio we observe only a 2% increase in wirelength over the full switch-boxes. These values greatly depend on the implementation and the data-flow used; however, they represent what can be achieved when typical designs are mapped to coarse-grain architecture.

Figure 5-12: The total length of the routings depending on the ratio between the number of Type 1 blocks and Type 2 blocks in switch-boxes (4) and (7).

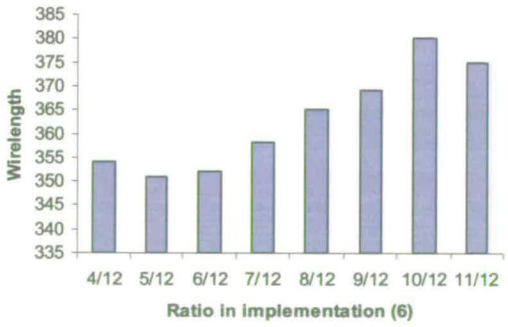


Figure 5-13: The total length of the routings depending on the ratio between the number of Type 1 blocks and Type 2 blocks in switch-box in (6).

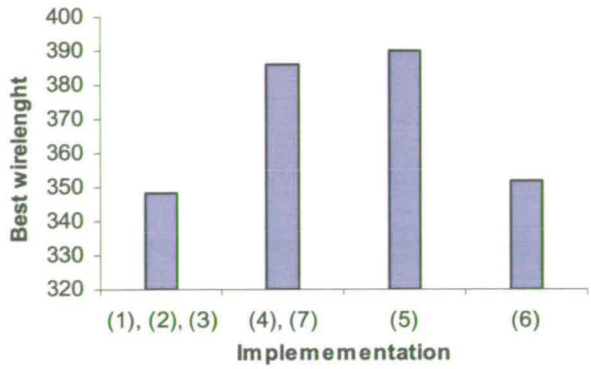


Figure 5-14: The total wirelength for each switch-box implementations. For (4), (7), (5) and (6) the ratio of Type1/Type2 with the lowest wirelength is chosen.

5.2.5. Analysis

From the above evaluations we can deduce that the compression of configuration data (as in (3) and (5)) only provides some area reductions for low widths of word-tracks. The use of 3-to-1 multiplexers (as in (2)) to implements full four-side switch blocks is inefficient as it increases the area, power and delays when compared to the use of tri-state buffers. Attractive results were achieved using switch-box with reduced directions ((4), (6) and (7)) when compared to full-directions switches.

The half-box based on tri-state buffers (implementation (4)) has low area, power consumption and delays but a large wirelength. Using 2-to-1 multiplexers (as in (6)) allows big improvements in routability at a price of a slightly larger area, longer delays and higher power consumption. Finally, the lowest area is achieved by combining multiplexers and tri-state buffers in the box (as in (7)) which give low-power consumption but slightly lower routability and longer delays (see Figure 5-15).

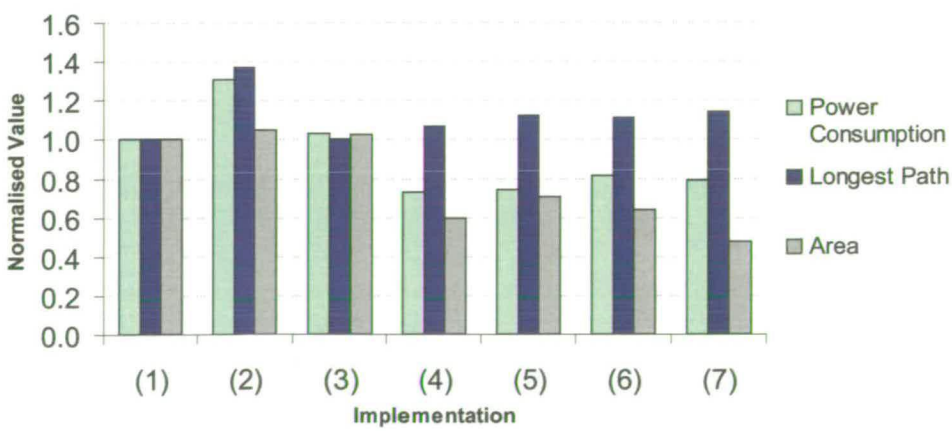


Figure 5-15: Comparison of the different designs in terms of power, area and delays.

5.3. Conclusion

It has been shown that the DSRA arrays can be further optimised to the application by tailoring the interconnects further to suit the application. In the given example, several directivities of the switch boxes were tested and the performance (area, power and timing) was measured. It was found that by making directivities of the programmable switches follow the intended data flow in the array, saving by up to 50% and 27% can be achieved in area and power, at the expense of only increasing the delays by 7%. On the circuit level, it was found that the lowest area and power were achieved by using a combination of 2-to-1 multiplexers and tri-state buffers in the 6W switch-point of the subset S-Box; the reason is that the total area of the S-Box depends on both the switching element used and the number of configuration bits required. The improvement in this type of S-Box comes at a price of increased delays and a lowered routability.

Chapter 6:

Reconfigurable Instruction Cells Array

In the previous chapters, the domain-specific reconfigurable arrays designed provided a good compromise between high-flexibility, high-power and high-area FPGAs on one side and low-flexibility and low-power ASICs on the other side. The DSRAs showed a throughput higher than FPGAs (and DSP processors), not very far from the level achieved in ASIC, while providing a good degree of flexibility. However, the two major drawbacks in the proposed DSRAs are, first, the long time required to design the DSRA itself according to the application, and second, the long design-time needed to map and program new algorithms on the array. As described earlier, programming the array occurs in a similar way to programming FPGAs using an HDL to represent netlists of programmed clusters. Ideally, a reconfigurable architecture would be programmable using a high-level (C/C++) programming language. Based on this, another limitation which emerges in DSRAs is the difficulty to automatically create an array tailored to the application starting from a high-level definition of the application, since the programming happens manually at low-level. Even though the

silicon-area usage of DSRAs was found to be lower than FPGA, it is still regarded as elevated when compared to the area occupied by ASICs or to the area of datapaths in typical CPU and DSP processors. This is mainly caused by the fact that 90% of the silicon is consumed by interconnects. Finally, the large number of configuration bits needed to configure a 'useful' section of the DSRA is too large (around 3000 bits) to permit dynamic re-configuration of that section, and hence it limits the possible rate of reconfigurability.

This chapter proposes a solution to overcome these limitations by changing the structure of the initial DSRA design. This is mainly carried out by moving from the previously described type of clusters into a cluster type that can directly execute assembly-like instructions commonly found in software implementations. Such clusters are called here *Instruction Cells* (ICs). The basic ideas presented in this section come from elaborations with other members of the research-group, mainly Ioannis Nousias along with Mark Milward and Ying Yi, who are working on the same project, namely the *Reconfigurable Instruction Cell Array* (RICA). I. Nousia's further work was to efficiently implement the data and program memory sub-systems along with coding of paths in the program memory using small foot-prints. M. Milward and Y. Yi were concentrating on optimised and advanced compilation software-tools.

This chapter introduces the instruction-cell based arrays and assesses the advantages/disadvantages gained by its structure. It also tries to evaluate the costs incurred by introducing programmability from high-level languages for what practically is a processor-like reconfigurable architecture.

6.1. Processor-like operation of a reconfigurable array

Assembly representations of programs – or more specifically the control and data flow graphs generated by compilers – can be regarded as an efficient low-level description of software and algorithms. This is especially useful due to the existence of compilers that convert high-level languages such as Java and C/C++ into assembly-instructions. In traditional and simplistic design of CPUs, the Arithmetic Logic Unit that performs the operations has typically only 2 inputs and one output, and according to the opcode it can perform operations like ADD, MUL or SUB to produces the output.

If each cluster in the DSRA can be made to execute one assembly instruction, then a computational datapath described in assembly-language can be simply executed in hardware by connecting the different 'instructions' together. An array containing such programmable clusters along with a mesh of reconfigurable interconnects can be configured to execute the required datapath. A full software program that includes branching and conditional operations would then be executed by dynamically re-programming the array to perform the different basic-blocks of the program. An instruction controller would then be responsible for handling

the branching operations. Making the DSRA clusters support assembly instruction would also be in theory an efficient way to reduce the overhead in interconnects (C-Boxes and S-Boxes) as the number of inputs and outputs is reduced to a minimum. It also allows the use of existing and mature compilers that would output suitable netlists of clusters to build datapaths from a high-level program representation.

6.1.1. Example of Instruction-Level Parallel Processing

The sample C code shown in Table 5-1 requires 19 cycles to execute on a typical sequential processor. However, if the same code is compiled for a VLIW DSPs, such as the TMS320C6x, then it would execute in 15 cycles, since the VLIW architecture would try to concurrently execute up to 8 independent instructions (6 ALUs and 2 multipliers are available) [113]. At 600MHz, 15 cycles translate to 25ns if we consider the ideal case where no instruction-pipeline needs to be filled. If 4 simultaneous multiplications and 4 memory accesses were permitted, then the number of cycles would reduce to 8. This is still high taking into account the simplicity of the code and when compared to what is achievable using hardware solutions like FPGAs. This speed limit is created by the presence of dependent instructions preventing the compiler from scheduling instructions in parallel and hence resulting in a high number of clock cycles. We can observe that if an architecture supports the mapping of both dependent and independent datapaths, then we could execute a big block of instructions in a single clock cycle without limitation.

Table 5-1: Example C-code and its assembled sequential and VLIW code compiled with level-2 optimizations

C Code	**Sequential ASM**

`b0 = in_mem[add+0];`
`b1 = in_mem[add+1];`
`b2 = in_mem[add+2];`
`b3 = in_mem[add+3];`
`e = b0 * f0 - b2 * f2;`
`f = b1 * f1 - b3 * f3;`
`out_mem[add+0]= e + f;`
`out_mem[add+1]= e - f;`
`out_mem[add+2]= f + 2*e;`
`out_mem[add+3]= f - e;`
	`LD [r3+0] →r11`
	`LD [r3+8] →r9`
	`MUL r11, r5→r11`
	`LD [r3+12]→r13`
	`LD [r3+4] →r3`
	`MUL r3, r6 →r6`
	`MUL r9, r7 →r5`
	`MUL r13, r8→r3`
	`SUB r11, r5→r5`
	`ADD r5, r5 →r7`
	`SUB r6, r3 →r3`
	`SUB r5, r3 →r8`
	`ADD r7, r3 →r7`
	`ADD r5, r3 →r6`
	`LD r8 →[r4+12]`
	`SUB r3, r5 →r3`
	`LD r6 →[r4+0]`
	`LD r3 →[r4+4]`
	`LD r7 →[r4+8]`

TMS320C6x VLIW ASM	

`LDH *+A4(2)→A7`
`LDH *+A4(6)→A3`
`LDH *+A4(4)→A0`
`LDH *A4→A5`
`MPY A7,B6→B5`
`MPY A3,B8→B6 || MPY A0,A8→A0`
`MPY A5,A6→A3`
`SUB B5,B6→B5`
`SUB A3,A0→A0 || EXT B5,16,16→B5`
`RET B3 || EXT A0,16,16→A0`
`MV B5→A3 || SUB B5,A0→B6`
`ADDAH A3,A0→A4|| STH B6→*+B4(6)`
`ADD B5,A0→B5 || STH A4→*+B4(4)`
`STH B5→*B4 || SUB A0,A3→A0`
`STH A0→*+B4(2)`	

15 Cycles (8 cycles if 4 MPY and 4 LD/ST are allowed)	**19 Cycles**

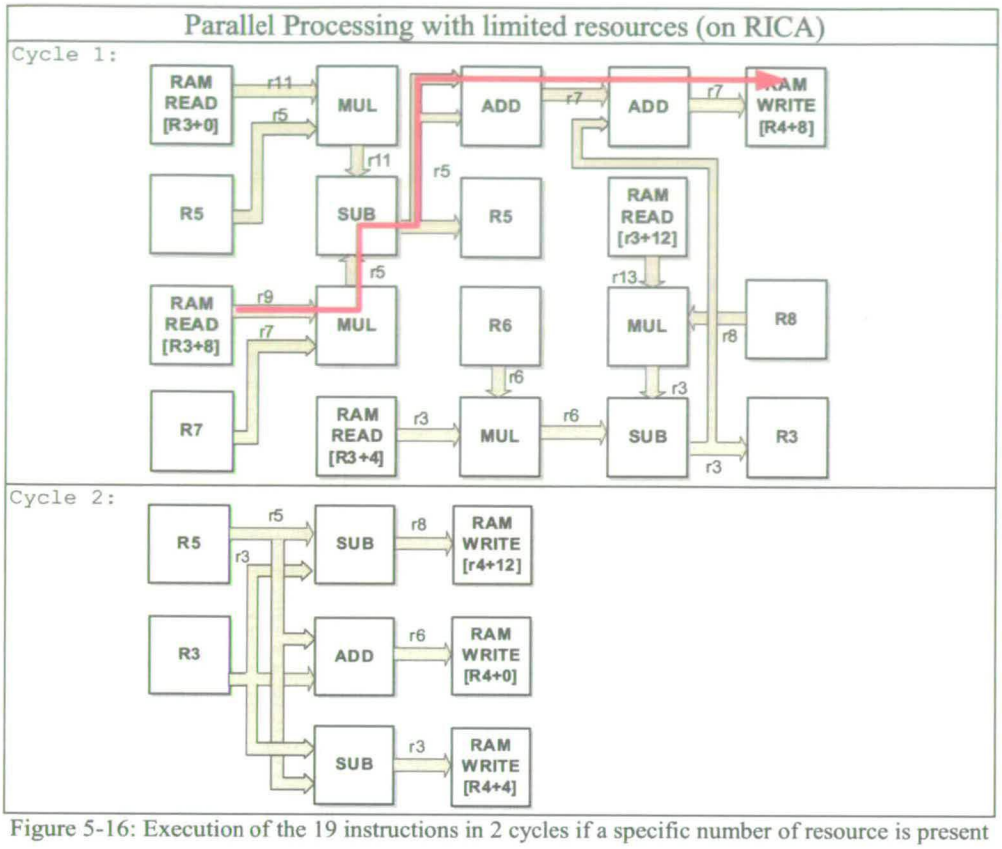


Figure 5-16: Execution of the 19 instructions in 2 cycles if a specific number of resource is present

We could easily execute the previous C code in only 2 cycles if the architecture provided 14 operational elements to perform 4xADD, 4xRAM, 4xMUL and 2xREG simultaneously, as shown in Figure 5-16. However, this would mean that the 4 RAM operations would access the main shared memory in parallel. This overcomes the Instruction Level Parallelism (ILP) limitation faced by VLIW processors and enables a higher degree of parallel processing. As shown in Cycle 1, the longest delay-path is equivalent to 2 RAM accesses, one multiplication and some simple arithmetic operations. This is not much longer than critical-paths in typical DSPs when compared to how many more instructions are executed in parallel during the same cycle. The 2 cycles translate to less than 15ns if typical (non-heavily constrained) DSRA delay values are used. Hence, an architecture that supports such an instruction arrangement might be able to achieve similar throughputs as VLIWs but at a lower clock frequency, depending on the type of computation.

6.1.2. Reconfigurable Core

The concept behind the RICA architecture is to provide a dynamically reconfigurable fabric that allows building such circuits – mapping the same circuit on the previous DSRA would require time-costly modifications and manual work that are difficult to automate. However, by providing DSRAs with clusters that can execute assembly-like instructions similar to the

ones in Figure 5-16, a straightforward design-flow resembling CPUs can be easily developed. The core elements of the RICA architecture are the Instruction Cells (ICs). Like in a DSRA, the ICs are interconnected together through a network of programmable switches to allow the creation of datapaths. In order to support the execution of large programs that do not fit into a single datapath, the configuration of the array should be allowed to change rapidly. Furthermore, to support conditional-executions that are found abundantly in typical software systems, the transition between the configuration-streams should be controlled by an instruction-controller in the same way it is done in normal processors. Similarly to CPU architectures, the configuration of the ICs and interconnects can be changed on every cycle to execute different blocks of instructions. Unlike CPUs and more like FPGAs, a circuit can also be mapped and executed for longer time (i.e. several cycles) if it is part of a loop. As shown in Figure 5-17, RICA can be implemented as a Harvard-architecture processor where the program-memory is separate from the data-memory. In the case of RICA, the processing data-path is a reconfigurable array of ICs and the program-memory contains the configuration bits (i.e. instructions) that control both the ICs and the switches inside the interconnects. Special ICs in the core are responsible for controlling the data and program memories.

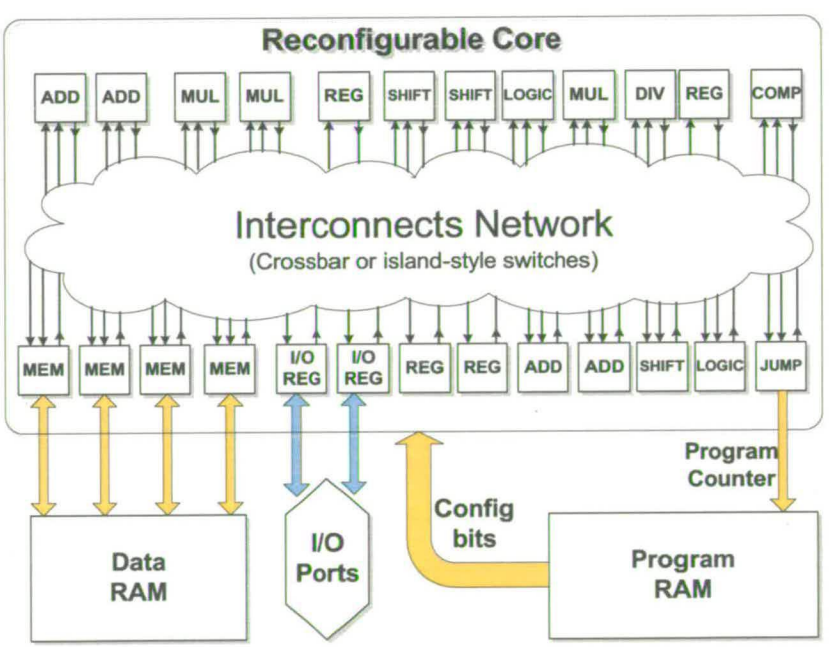


Figure 5-17: Harvard-like structure of the RICA with reconfigurable core as instruction-cells and programmable interconnects

Although the RICA architecture is similar to CPUs when using program controllers and dapaths, the use of an IC-based reconfigurable core as a data-path gives important advantages over DSP and VLIWs, such as more support for parallel processing. A reconfigurable core can execute a block containing both independent and dependent assembly instructions in the same clock cycle, which prevents the dependent instructions from limiting the amount of ILP

in the program. Other improvements over DSP architectures include reduced memory access by eliminating the centralized register-file and the use of distributed memory elements to allow parallel register access.

In a similar way to DSRAs, the characteristics of the reconfigurable RICA core are fully customizable at design-time and can be set according to the application's requirements. This includes options such as the bitwidth of the system, which can be set to anything between 4-bits and 64-bits, and the flexibility of the array, which is set by the choice of ICs and interconnects deployed. These parameters also affect the extent of parallelism that can be achieved and device characteristics such as area, maximum throughput and power-consumption. Once a chip containing a RICA core has been fabricated, the system can be easily reprogrammed to execute any code in a similar way to a processor.

6.2. Hardware design

6.2.1. Instruction Cells

In contrast to other reconfigurable architectures (see Chapter 2), the IC-array in RICA is heterogeneous and each cell is limited to a small number of operations as listed in Table 5-2. This allows us to increase the overall cell count to do more parallel computations, since the overhead of adding such small cells is merely related to the extra area occupied by the interconnects. The use of heterogeneous cells also permits tailoring the array to the application domain by adding extra ICs for frequent operations. Each IC can have only one instruction mapped to it. In a similar way to assembly instructions, all cells have only 2 inputs and 1 output – this facilitates creating a more efficient interconnects structure and reduces the number of configuration bits needed. The cells initially developed support the standard instruction-sets found in 32-bit GPPs like the OpenRISC [117] and ARM7 [115]. Hence, with such an arrangement, RICA could even be made binary compatible with any existing GPP/DSP system.

As shown in Table 5-2, registers memory-elements are defined as standard instruction-cells distributed throughout the array, which allows them to operate independently to increase degree of parallel processing. As seen in the previous example, to program RICA the assembly code of a software is sliced into blocks of instructions that are executed in a single clock cycle. Typically, these instructions – that were originally generated for a sequential GPP – would include access to registers for the temporary storage of intermediate results; in the case of the RICA these read/write operations are simply transformed into wires to reduce the registers-use. By using this arrangement of registers RICA also offers a programmable degree of pipelining operations and hence it easily permits breaking up long combinatorial computations into several clock cycles.

Special ICs include the JUMP cell which acts as an instruction-controller responsible for managing the program counter and the interface to the program-memory. The interface with the data-memory is provided by the MEM cells; a number of these cells is available to allow simultaneous read and write from multiple memory locations during the same clock cycle. This is achieved by using multiple memory banks to form the data-memory and by clocking it at a higher speed than the reconfigurable core; this is possible since the core needs a relatively low clock frequency typically equivalent to around 40MHz (see description of the CLK_DIV cell for the clock equivalence). Furthermore, some special REG ICs are mapped as I/O ports to allow interfacing with the external environment.

This is only an initial division and the scope of the operations of the cells can be expanded in the future. It is also possible to have a large IC supporting the typical operation of an ALU in a GPP: arithmetic, shifting, logic and memory.

Table 5-2: Possible Instruction Cells and their operations

Instruction Cell	Supported Operations
ADD	Addition, Subtraction
MUL	Multiplication (Signed and Unsigned)
DIV	Divisions (Signed and Unsigned)
REG	Registers
I/O REG	Register with access to external I/O ports
MEM	Read/Write from Data Memory
SHIFT	Shifting operation
LOGIC	Logic operation (XOR, AND, OR, etc.)
COMP	Data comparison
MUX	Multiplexer
JUMP	Branches (and sequencer functionality)
ALU	Full CPU-like arithmetic logic unit

Data signals that can be routed between two cells or stored in registers consists of N-bit data and 1 carry bit, e.g., if a 32-bit RICA is designed, the signals would be 33-bit wide with one carry bit. Using this carry signal we can cascade several cells to achieve high precisions computations, such as 64-bit additions or multiplications. See Appendix A for the details of the instruction cells in the sample RICA.

ADD

This cell supports addition and subtraction operations. There are 2 input data and one output signal. In the configuration we can select which bit-precision the cell should use (e.g. 8-bit, 16-bit or 32-bit mode). As will be seen in Section 7.1.2, this cell can also be configured to support complex addition/subtraction; in this case the input data is split between the real and imaginary parts (e.g. a 32-bit RICA would have a 16-bit imaginary part and a 16-bit real part).

MUL

This cell support signed and unsigned multiplication. Similar to the ADD cell, it can also support complex multiplication and cascading to achieve higher precisions.

RMEM

This cell gets as input an address and an offset and reads the content of the Data RAM at the required location. The reading from the Data RAM currently takes place each time the address at the input of the cell changes at any time during the step. This is necessary since in situations where we are accessing an address pointed at by a variable in memory (i.e. a *pointer*) a cascade of two RMEM is created, and hence the second RMEM should be reading the data from the memory only when the first RMEM has finished outputting the required address. In the future, time tags can be introduced to detect when (during the execution clock cycle) the address and offset are ready to start fetching data from the Data RAM; the computation of the time tag can be done by the compiler [129].

WMEM

This cell gets data and writes it in the Data RAM. The data to be written is latched at the end of the cycle and is written in the next step that contains any read operation from the Data RAM.

REG and I/O REG

The REG cells replace the register file found in a processor, with the difference that the registers are distributed and accessed independently; hence they consume less energy since there is no need to use a large multiplexer to address them. Each register can have several data banks inside it. In the sample array described below in Section 6.4 it was decided to use 2 banks for every register, as this helps optimising leaf functions (functions that don't call any other functions) by removing the need for saving the used registers in the stack.

Another version of these REG cell is an I/O REG cell, which represent an *N*-bit dataport; when writing data to the port it would be transferred to the chip's pins, and when reading the register's content it would be coming from the chip's pins. The I/O REG has to be configured as input or output.

DIV

This cell support signed or unsigned division.

LOGIC

The LOGIC cell can perform standard bit-operations such as AND, OR, NAND, NOR, XOR, NOT, as well as bit-reversion and 2's complement negating.

SHIFT

This cell can perform logical and arithmetic left/right shifting.

COMP

The COMP cell compares two inputs and output is the result of the comparison generated as a data signal. This output should be routed to either the MUX or JUMP cell.

MUX

This cell receives 3 inputs: Two data signals in_1 and in_2 and the result of the comparison coming from a COMP cell. According to the result of the comparison it either routes in_1 or in_2 to the output. Hence it acts as a multiplexer, if seen from the hardware point of view, or a conditional-move operation if seen from a software point of view.

JUMP

The JUMP cell acts as the instruction-controller and manages the *Program Counter*. The program counter is given to the Program Memory controller to retrieve the configuration of the cell for the current steps. During the execution of a step, the JUMP cell computes the value of the next program counter so that the configuration of the next step would be ready when needed. The computation of the next location can be conditional by using the output of a COMP cell, and hence achieving conditional branching in RICA.

CLK_DIV

The CLK_DIV is responsible for 'dividing the global clock' and setting the period for which a single configuration should be running for. This is needed in RICA since there is a big variation of delays between different steps of a program. This variation is created by the fact that we can execute dependent instructions connected together in series, and hence, every circuit has its own critical-path delay. If this cell was to be omitted, then the maximum operating frequency of RICA would be limited to the largest longest-path delay in all the steps of the whole program. With the CLK_DIV cell it is possible control the execution time needed for each step, and hence make this delay only limited to the longest-path delay in the step itself. The configuration of CLK_DIV is computed at compile-time according extracted worst-case values.

The CLK_DIV outputs an *enable* signal that goes to all the WMEM, REG and JUMP cells (the only sequential cells in RICA) to signal the end of the time allocated to the step.

6.2.2. Interconnects

Interconnects allow routing the signals between the instruction cells. As described earlier, the signals are composed of *N*-bit data and a carry bit (generated in adders or multipliers). Two interconnects schemes were investigates for RICA: Interconnects based on crossbar multiplexers and island-style routing boxes.

Table 5-3: Comparison between cross-bar and island-style interconnects

Interconnects (sample array with 64 cells)	Area on 0.13µm (μm^2)	Number of cfg bits	Delay of one connection (output-input, ignoring wire capacitance)
Multiplexers	1,640,495	498	0.7 ns
Island-style	576,062	678	Variable, average of 5 s-boxes is 2.0 ns

The programmable switches should perform directional connections between the output and input ports the cells. The design of the interconnects should take into account that each instruction-cell has only one output and up to 3 inputs and that in no case will the output of a cell be looped back to one of its inputs (to avoid combinatorial loops).

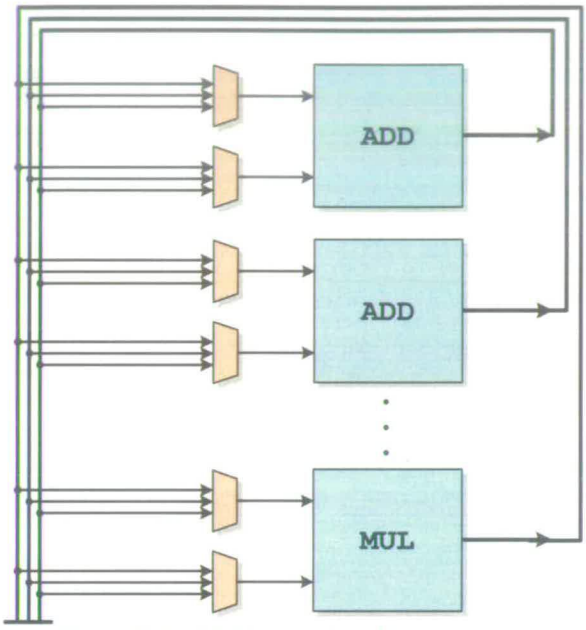


Figure 5-18: Multiplexers based interconnects

The multiplexer-based crossbar is shown in Figure 5-18. It is based on a simple design where each input of each cell has a multiplexer to select which cell's output should be routed in. In a typical array (see the sample array in Section 6.4) there are about 64 cells, with around 60 cells having outputs (some cells such as WMEM have no outputs), hence the 32-bit multiplexer would be of size 59-to-1 (the cell itself is not used). Such a multiplexer is very big and consumes a large area as shown in Table 5-3. The cells in the sample array have 83 inputs ports each requiring such a multiplexer. In the sample array, multiplexers would consume around 68% of the array area, i.e. the area of the interconnects is 2.1 times the silicon area of computational cells themselves. The delay associated with the multiplexer to route the signal from the output of one cell to the input of another is around 0.7 ns, which is around 20% less than the delay required for an ADD cell (0.9 ns). The delay is formed by passing through 3 levels of 4-to-1 multiplexers from the standard-cell library. It should be noted that this value ignores the line capacitance associated with the wire and that such a crossbar scheme would result in long wires.

Another problem with multiplexer-based designs is that the interconnect's area grows rapidly when the number cells increases. Figure 5-19 shows the synthesised area of a multiplexer for different number of input pins. As can be seen there is a rapid change in area for N=32 after which the area grows somewhat linearly. This linear increase has an exponential effect on the

total area occupied by all multiplexers when the number of cells is increased, as shown in Figure 5-20, The exponential relationship is due to the fact that for each new cell added to the array we need to increase the size of the multiplexers of all the existing cells. Hence, multiplexer based interconnects limit the scalability of the architecture.

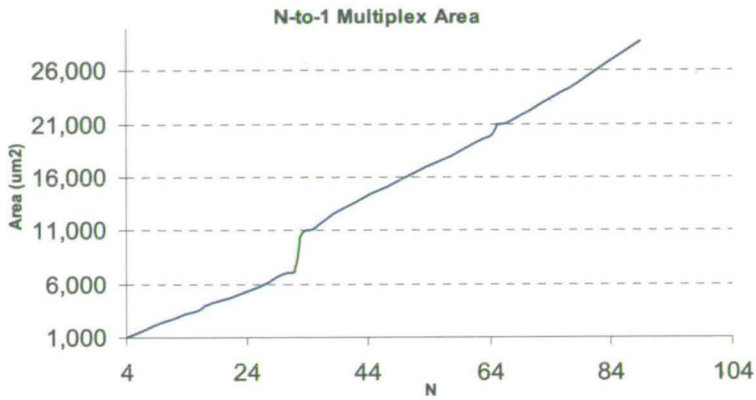


Figure 5-19: Silicon area of *N*-to-1 multiplexer

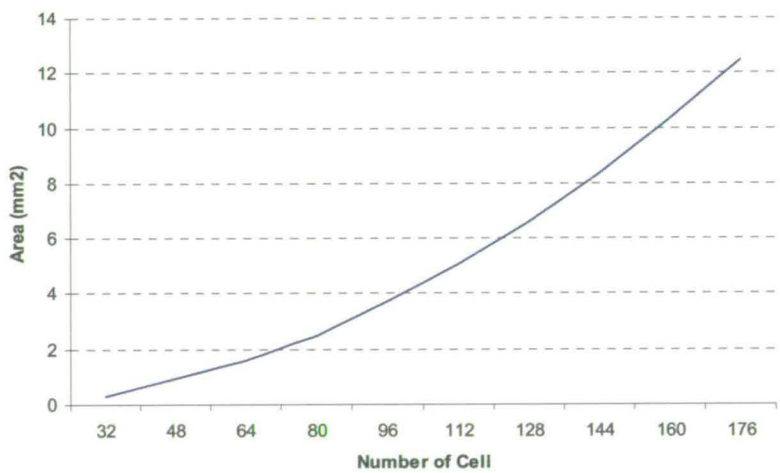


Figure 5-20: Exponential increase of silicon area with number of cells when using multiplexers

The second interconnect scheme considered is the island-style shown in Figure 5-21 and Figure 5-22. Each cell is surrounded by 4 routing multiplexers, one for each side. The signal tracks used are unidirectional, and on each side there is one input and one output. The multiplexer controls the output signal, and according to its configuration it can route signals that are coming in from other directions to its output. Each multiplexer also receives the output of the current cell to allow routing it to other cells. Furthermore, each input pin of the instruction-cell has a 4-to-1 multiplexer to select which of the four sides should be routed from outside of the box. As can be seen from Table 5-3, the overall area of these routing elements is 64% smaller than the crossbar multiplexers. In addition, they are much more scalable and make it realistic to implement arrays with more than 64 cells. On the downside,

the number of configuration bits required is increased by 36%. The delay is also increased and becomes dependent on the routing of the signal and the number of s-boxes it passes through. However, the value given does not include wire delays, which in this case should be much less than the crossbar version, as the metal wires are greatly reduced due to the increased locality.

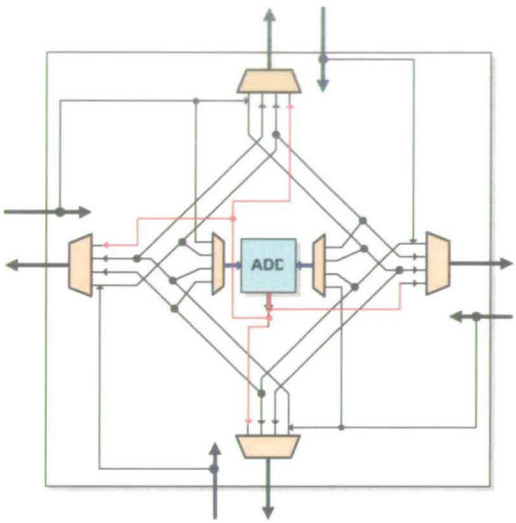


Figure 5-21: Configurable switches around each cell to form an interconnects-box for the island-style mesh.

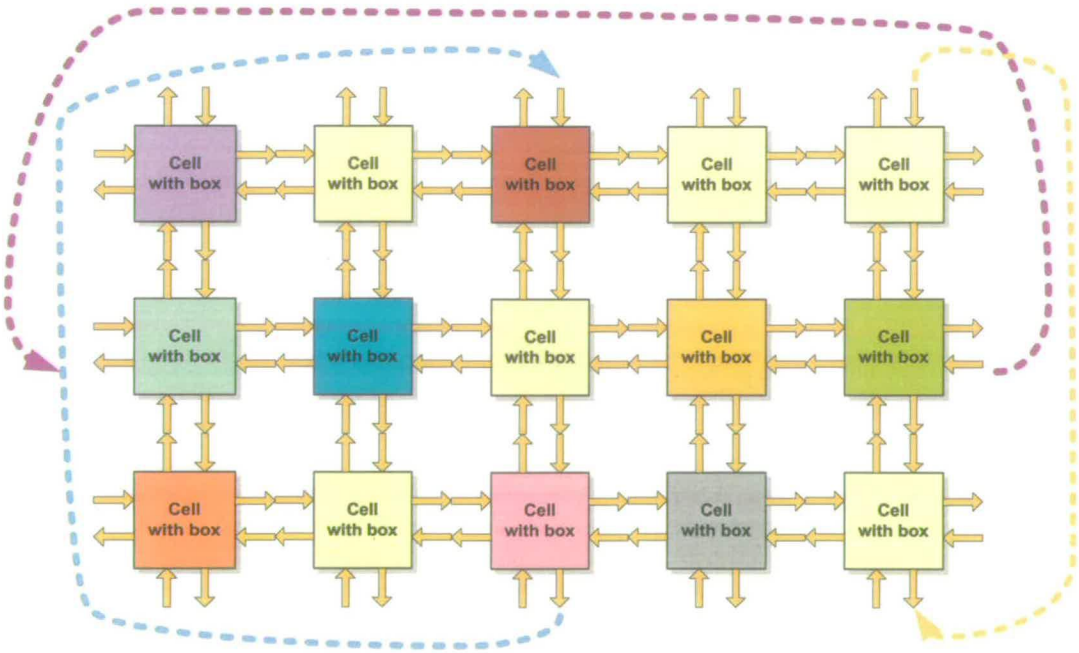


Figure 5-22: Mesh of island-style interconnects with torodial interconnects

Another effect of using the island-style scheme is that the correlation in the configuration bits of different steps is low. In the case of the crossbar, a cell that is active would have its multiplexer active as well; however, in the case of the island-style mesh a cell might be

inactive in the specific step but its associated s-box might be used to route a signal belonging to a different cell. The effect of this observation has to be taken into account in the future if a compression scheme based on temporal or spatial redundancy is to be used on the configuration bitstream.

6.2.3. Data Memory interfaces

The RICA array can have a number of Data RAM access cells, such as 4x RMEM. When a program is compiled for RICA, the compiler assumes that these RMEM cells operate in parallel. This can be physically achieved by using different and independent memory banks for each of these cells. However this solution would require the compiler and scheduler to know in which memory bank each location is stored. Another solution is to use memory banks that are time-multiplexed between the 4 RMEM cells so that only one cell accesses one of these memory banks at any one time. As described earlier, RMEM acts a combinatorial cell and the data is read from memory each time the input address to RMEM changes. The time-multiplexing circuitry has to keep cycling between all the shared RMEM cells to check which one had an address change so that the data gets read.

6.2.4. Program Memory implementations

One drawback of the proposed cells and interconnects is related to the number of bits required to store their configuration, which is in the order of 500-800 for the tested case using multiplexers interconnects. Since the configuration of the cells is changed every step in a program, we would need to store the configuration of every cell in every step. For example, the code for an MPEG-2 Layer III audio decoder takes around 1,500 steps. This amounts to around 1,500 x 700 ≈ 1 MBit of program memory. This is quite large considering the fact that the same code fits into 440 kBit of memory when compiled for a CPU like ARM or OpenRISC.

This high program memory usage affects the overall power consumption of the design and might offset any power saving achieved using the datapath. Fortunately, on average only around 12 cells are active in any step in the largest benchmark MP3 program from Section 7.2 and hence the 1MBit of data contains a lot of redundant information like *nop* (no-operation) configuration. The existence of this redundant information can be used to compress this configuration memory. Several compression techniques were investigated, and an ongoing project looking at reducing the amount of configuration data using distributed configuration memory showed promising results. The compression of the configuration memory is beyond the scope of this document. In this document we implemented only programs small enough to fit uncompressed in the available memory (See Section 6.4.2).

6.3. Design-Tools for RICA

An automatic tool flow has been developed for the generation of RICA arrays based on the initial tools for generating DSRAs. In a similar way to DSRAs, the tools take the characteristics of the required array and generate a synthesizable RTL definition of a RICA core that can be used in standard SoC software-flow for verification, synthesis and layout. These characteristics include the number of cells, type of interconnects, placements of the cells in the array and number of rows and columns. If the RICA is implemented using crossbar multiplexers, then it would be defined by the tools as an array with a single row.

The main advantage of RICA over DSRAs and FPGAs is its ease of programming. The overall tool-flow needed for this is shown in Figure 5-23. The use of Instruction-Cells greatly simplifies the overall design effort needed to map high-level programs to the RICA architecture through a CPU-like programming flow. First, a compiler is required to transform the input high-level languages, such as C/C++ or Java, into instruction-cells description. The second step schedules the instructions, according their dependencies, for execution into multiple steps on RICA. The final step generates the configuration of interconnects and cells for implementing the desired steps.

It was decided to use the open-source standard GNU C Compiler (*gcc*) [118] as the front-end compiler for RICA, since it is highly customisable and currently the best available open-source compiler. GCC supports different language inputs amongst them C/C++, Java, Fortran and Ada. In the ideal case, the *gcc* package would be responsible for the first two steps described earlier, i.e. compilation and instruction scheduling. This would allow achieving RICA-specific optimisation at compile time by making *gcc* aware of the resources available on RICA. However, at the start time of the project, the *gcc* version available had limited support for parallel instruction execution. Independent instructions could be identified by the compiler for parallel scheduling, however, too much work was required for supporting blocks of both dependent and independent instructions.

Hence, it was decided to modify *gcc* to generate instructions for the RICA cells in a serial format; the compilation is performed by *gcc* with the assumption that the created instructions will be executed in sequence. This RICA-specific assembly, which describes which ICs need to be used, is then processed by the RICA *scheduler* to create a sequence of netlists each containing a block of instructions that are executed in one clock cycle. The netlists represent the different steps that have to be executed in sequence, with each step containing several instructions that are to be executed in parallel and/or series.

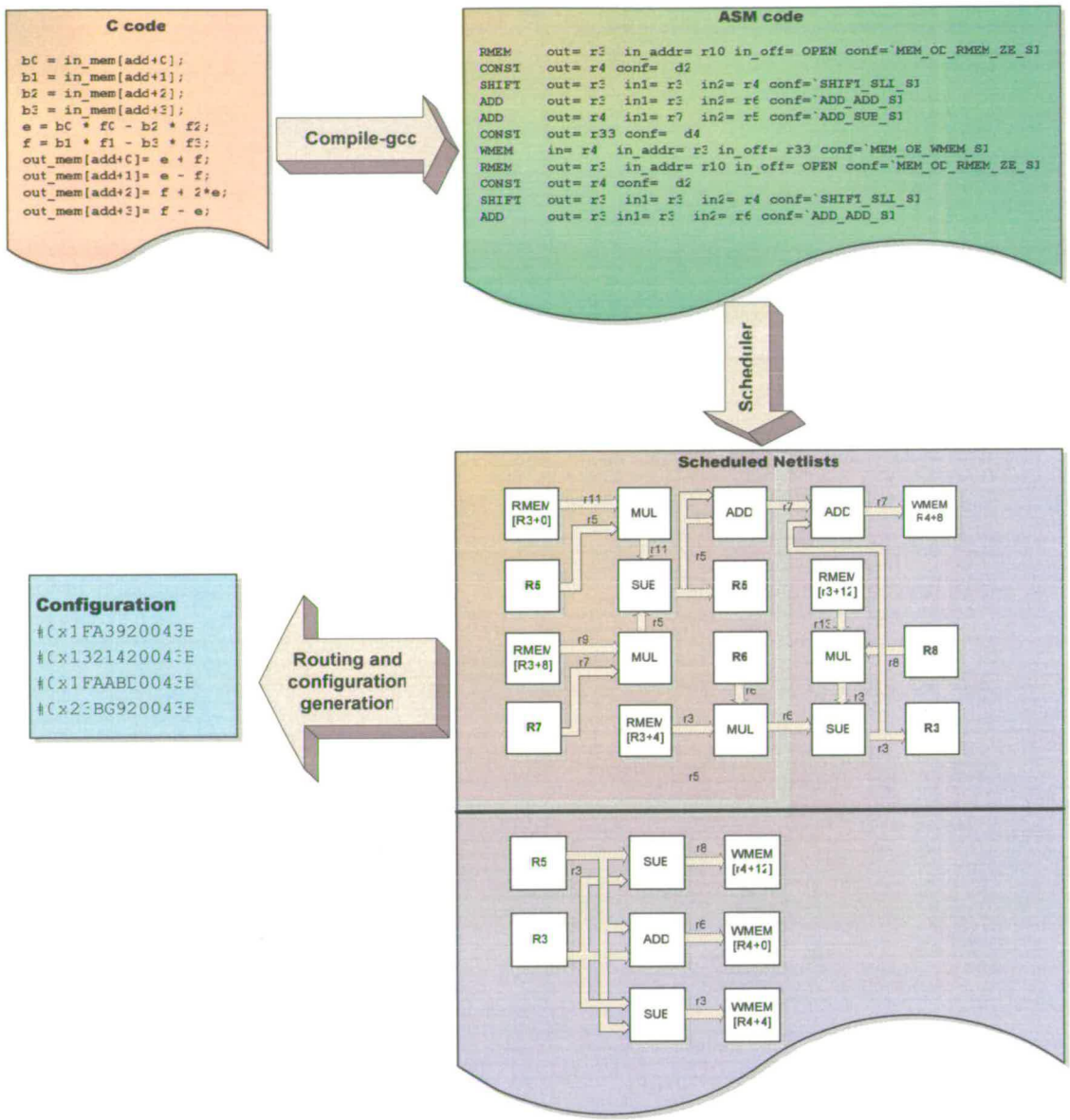


Figure 5-23: Design-software tool-flow for RICA.

This splitting is not efficient, as *gcc* would be performing register allocation internally and passing it to the scheduler. The scheduler has then to execute the instruction scheduling while being restricted in using the registers previously allocated by *gcc* for each basic-block. The effect of this is that some basic blocks would be split in more steps than required, which is due to the unavailability of temporary registers.

The simple scheduling algorithm used takes into account IC resources, interconnects resources and timing constraints in the array. It tries to have the highest program throughput by ensuring that the maximum number of ICs is occupied and that at the same time the longest-path delay is reduced to a minimum. The instruction scheduling is performed on each basic-block separately. The first step in the scheduling is to convert the *move* instructions and all register operations found in the assembly into wire connections between ICs. This implies

that the register allocation carried out by *gcc* is partially lost. However, the scheduler has to ensure that no register is used in the resulting steps other than the ones already in use by the original basic-block. The scheduling algorithm then executes inside a loop that tries to find which instruction has to be scheduled next. A cost is computed for each unscheduled instructions which takes into account the following 3 constraints:

- The resources availability
- The availability of temporary registers
- The longest-path delay in the resulting step

The algorithm then selects the *cheapest* cell to be scheduled, and the loop is started again. If no instruction could be scheduled, the algorithm will create a new sub-step of the current step and tries scheduling again. The use of these 3 constraints (they can be used with different weights) makes the scheduler try to minimise the longest-path by executing more parallel and independent instructions, while restricting to the available registers and resources.

This simple algorithm works in most cases, however, it fails in some situations due to the lack of registers in the basic-block. As described earlier, this is caused by the fact that *gcc* tries to minimise register usage inside the block. In such cases, a manual modification was needed to make the assembly output from gcc pass the scheduling. During the course of the work a new version of gcc was released (4.0 and beyond) which improved support for parallel instruction issuing. An ongoing project is now responsible for integrating a better quality scheduler into *gcc* for RICA, so that such register allocation problem can be avoided. However, the simple scheduler was enough to test the performance of RICA when running simple programs as described in the next section.

After the generation of the netlists, or steps, the configuration data for RICA is created. If island-style interconnects are used, then the configuration of the multiplexers has to be computed to make the connections between the cells. As is the case with DSRAs, this step can be done using VPR [57] tailored to the routing structure. All the cases tested in the performance evaluation (see below) were routable using VPR. However, in future versions of the scheduler, the routability of the designs should be included as a constraint when calculating the cost of scheduling an instruction. If the crossbar interconnects are used, VPR is not needed and the configuration can be generated directly.

6.4. Performance evaluation of sample RICA

The sample RICA array chosen for comparison contains the cells listed in Table 5-4 interconnected using multiplexer-based switches. The IC selection was done manually as it was adequate for general applications – as described earlier, although other combinations can provide better performance depending on the application. These 32-bit cells provide the same basic functionality as a general 32-bit DSP such as the ARM7. With the selected type of

interconnects and ICs the reconfigurable core requires a 738-bit wide instruction word. The array was implemented using a UMC 0.13μm technology.

Table 5-4: Instruction Cells in the sample array

Cell	Count
ADD	4
MUL	4
REG	32
SHIFT	2
DIV	1

Cell	Count
LOGIC	2
COMP	1
JUMP	1
MEM	8

6.4.1. Comparison with DSRA

An 8-point 1-D DCT was implemented on RICA for comparison purposes with the DCT mapped on the DSRA using Distributed Arithmetic from Chapter 4. It should be noted that RICA has been implemented using 0.13μm, while the previous DSRA use 0.18μm; hence the performance values shown in Table 5-5 had to be scaled from the ones in Chapter 4[1]. Also, the DCT on the DSRA is a 12-bit DCT, while the sample RICA used is 32-bit. A 32-bit DCT on the DSRA would require 32 cycles to finish (since the DA implementation is bit-serial) and would need larger accumulators to store 32-bit results. The execution time shown in Table 5-5 for RICA include just the time needed to run the DCT and no other operation such as memory initialisation (which is included later on when RICA is compared with processors).

Table 5-5: Comparison of the 8-points DCT on RICA and DSRA

	RICA	DSRA
Area (mm^2)	2.1	-
Code size (bytes)	5,621 (Config Stream, FF)	2,460 (Program Memory, SRAM)
Total area estimate (mm^2)	**2.27**	**0.096**
Minimum execution time (μs)	0.08	0.13
Energy for 1 DCTs (nJ)	4.1	88

The large differences in the measured performance charachteristics show the difficulty that exists when comparing hardware and software implementations, as each of the implementations has been tuned for a specific optimisation. It can be seen that the DSRA is around 20 times smaller, while it consumes around 20 more energy. Such results are expected, since the energy was measured while running the simulation at the highest possible frequency and the contribution of leakage power (which is proportional to the area) was kept to a minimum. It can be seen that bit-serial DCT implementation on the DSRA is effective in reducing the area at the expense of an increased switching activity and power consumption. A bit-parallel DSRA would have been more appropriate for the purpose of this comparison.

[1] The scaling factors were found by forming an average of the ratios between the datasheets of the two UMC technologies; this was done for area, delays and energy.

Hence, no exact figure can be extracted on the costs of programmability that was brought by using instruction-cells over programmable clusters. However, it seems as if they both give similar performance.

6.4.2. Comparison with DSP Processors

The sample RICA was compared to the following DSP architectures: The simple OpenRISC CPU [117] implemented on UMC 0.13μm technology, the ARM7-TDMI-S [115] again on 0.13μm technology, the TI C55X [119] 2-way datapath low-power DSP and the powerful TI64X 8-way VLIW [113]. The benchmarks are mainly based on TI's benchmarks for the TI C64X. All the benchmarks are direct unoptimised C representations of the algorithms – all optimisations are left for the C compilers (Level-3). The compiler used for the RICA did not include any advanced techniques like predications or the use of rotating register as compiled provided by TI does. All benchmarks include memory transfers, stack control and function's prologue and epilogue and hence they show a representative evaluation of the architecture's performance.

Program size issue

In the results shown here, only the datapath energy consumption is measured for the execution of the complete benchmark and compared to the architectures. It is important to note here that the power consumption of the program memory is not included in the evaluation. In the presented data, the programs used for RICA are raw and have not been compressed, which means that they are abundant in redundant *zero* configurations. Formatting the program memory in a similar way to VLIWs where the end of each step is marked using a tag can be easily applied to reduce the program size. However, such a formatting would not bring the program down to the same size as the VLIWs, since in RICA the array is heterogeneous and the location of every instruction has to be hard coded. Work has been done in evaluating the distribution of the program-memory elements to each cell which helps in removing a section of the redundant information. However, this is beyond the scope of this thesis. More work has been by other members of the group on compressing the program as part of a path-encoding scheme useful when used with island-style interconnects [128].

Measurements

For the RICA and OpenRISC the power and area were found using post-layout simulations. The ARM7 datasheet [115] provides power and area values of the core in 0.13um technology, while [120] and [114] allows us to estimate the power consumption of just the datapaths in the TI C64x and TI C55x. All these power estimations were measured at 1.2V operating voltage. The area of the datapath in the TI C64x was estimated using scaling from the published die-photo [111] knowing that the whole chip has 64M transistors (no cache memory was

included). No area information was available for the C55x. Table 5-7 also include variations in program size, as they differ for each architecture and compiler technology used. The size of the data-RAM is the same for all processors, and hence it is not included in the comparison. The Dhrystone benchmark, which today has become an outdated measurement, is included here for reference. As shown in Table 5-7., the fact that the Dhrystone takes more cycles to run on the highly pipelined TI DSPs than on the ARM7 shows how specific a benchmark it is. The fact that the Dhrystone compuation requires a large number of non-predictable brach-operations forces highly pipelined DSPs to frequently flush the instruction execution pipeline and hence waste time. Using it as a single benchmark hides a lot of the speedups achieved in modern media and DSP processors.

Results

The results are listed in Table 5-5 Table 5-6, Table 5-7 and shown in Figure 5-24 and Figure 5-25.

Table 5-6: Comparison of datapath area on 0.13um of CPUs excluding variations in program memory.

	RICA	OR32	ARM7	C55x	C64x
Datapath Area (mm^2)	1.90	0.25	0.32	N/A	2.01

Table 5-7: Comparing RICA with other processor, low-power DSP and VLIWs using benchmarks.

	RICA				OpenRISC CPU (on UMC 0.13μm) - 112MHz				ARM7-DTMI-S (Syn. on 0.13μm) - 110 MHz			
	CLK_DIV Cycles	Min Execution Time (us)	Raw Code (bytes)	Energy per Op (nJ)	Cycles	Min Execution Time (us)	Code size (bytes)	Energy per Op (nJ)	Cycles	Min Execution Time (us)	Code size (bytes)	Energy per Op (nJ)
1-D DCT	43	0.12	993	**4.7**	102	0.91	402	**10.2**	104	0.95	406	**9.36**
2-D DCT	1351	3.01	1785	**159.3**	4972	44.39	516	**497**	3760	34.18	508	**338**
Viterbi	1838	7.78	1286	**218.3**	9032	80.64	308	**903**	8803	80.03	316	**792**
IIR	120	0.17	755	**16.33**	180	1.61	510	**18**	176	1.60	464	**15.8**
Min Error	5164	11.10	1070	**620.1**	9073	81.01	442	**907**	8908	80.98	412	**802**
Dhrystone	798	1.12	1289	**52.57**	711	6.35	870	**71.1**	712	6.47	912	**64.1**

	TI C64x 8-ways VLIW - 600MHz				TI C55x 2-way low-power DSP - 300 MHz			
	Cycles	Min Execution Time (us)	Code size (bytes)	Energy per Op (nJ)	Cycles	Min Execution Time (us)	Code size (bytes)	Energy per Op (nJ)
1-D DCT	68	0.11	316	**34.68**	104	0.35	451	**26**
2-D DCT	1763	2.94	588	**899.1**	2300	7.67	655	**575**
Viterbi	3120	5.20	664	**1591**	3980	13.27	262	**995**
IIR	39	0.07	160	**19.89**	139	0.46	436	**34.8**
Min Error	1320	7.20	952	**673.2**	7479	24.93	380	**1870**
Dhrystone	928	1.55	424	**473.3**	916	3.05	1021	**229**

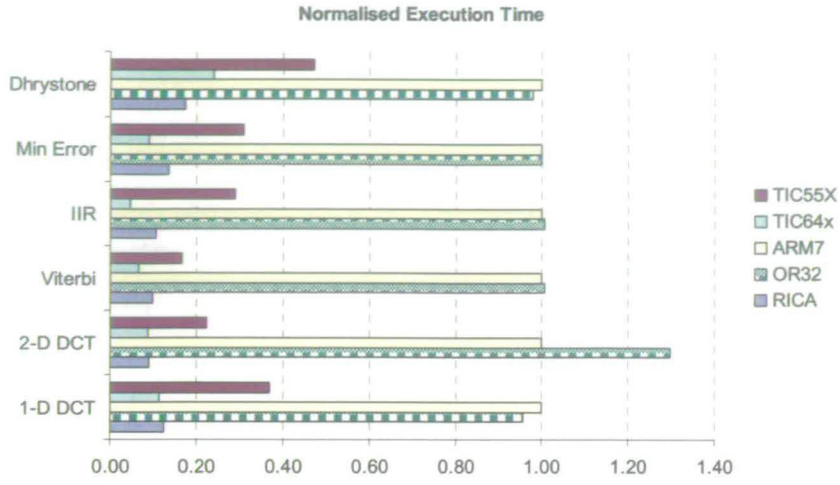


Figure 5-24: Normalised execution time graph of the benchmarks on RICA and other architectures

From the tables, we can see that for all the benchmarks we achieve better performance on RICA that on the conventional OR32 and ARM7 CPUs: We obtain around 1-3.6 times less energy consumption while achieving around 5-8 times higher maximum throughput. Due to the increase in program size memory and the increase in the datapath area, the power and throughput improvements come at the cost of an area increase of around 7 times in area. A large part of the power reductions achieved over the four DSP systems are savings gained by eliminating the registers-file and having distributed registers.

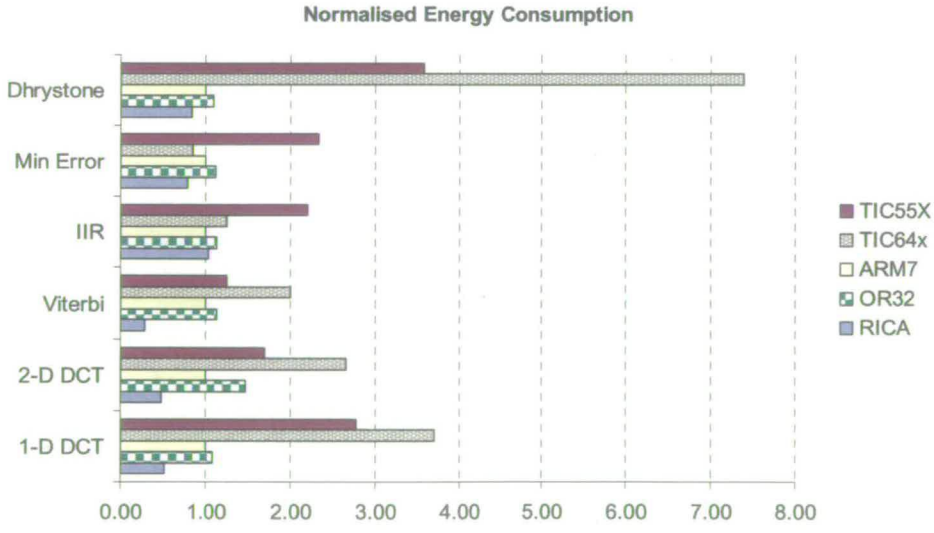


Figure 5-25: Normalised energy consumption graph of the benchmarks on RICA and other architectures

When compared to the low-power C55X DSP, RICA achieves a promising reduction in energy consumption between 2 to 6 times while achieving a throughput of up to 3 times higher. RICA achieves similar timing performances to the VLIW for applications containing

significant datapath operations like DCT, while faster operation is seen for Dhrystone. For benchmarks containing a large number of independent blocks and control parts (i.e. small loops and comparisons) like Minimum Error, RICA is around 50% slower than the 600MHz VLIW – this is expected as the TI compiler can optimise such code by using techniques such as predication (i.e. conditional execution) in a better way than *gcc*. For the Viterbi and IIR, RICA was around 20%-30% slower with the bottleneck being memory access. However, for the case of the Viterbi, the *gcc* compiler was able to correctly identify the use of multiplexers which improved the operating speeds and reduced branching. It should also be noted that RICA is built from synthesisable standard-cell libraries while the circuits in the VLIW have been manually laidout to achieve the 600MHz operating frequency. In terms of energy, around 6 times less power is consumed for DCT, Viterbi and Dhrystone; this is caused by the fact that on RICA less time is spent with large ALUs idel but consuming pouwer. The power reductions for the Minimum Error and IIR benchmarks were lower at around 17%. In terms of area, the datapaths of the RICA and VLIW are similar.

6.5. Reconfigurability overhead

As expected, the relative area occupancy of interconnects varies depending on the interconnects type used (shown in Figure 5-26) which represents the average values measured for the different application. The multiplexer-based interconnects occupy 78% of total core area; this is quite a large overhead, however, it is still less than the 90% figure found in normal FPGAs and the DSRAs. If island-style s-box interconnects are used, then the total contribution of the interconnects to the area comes down to around 40%.

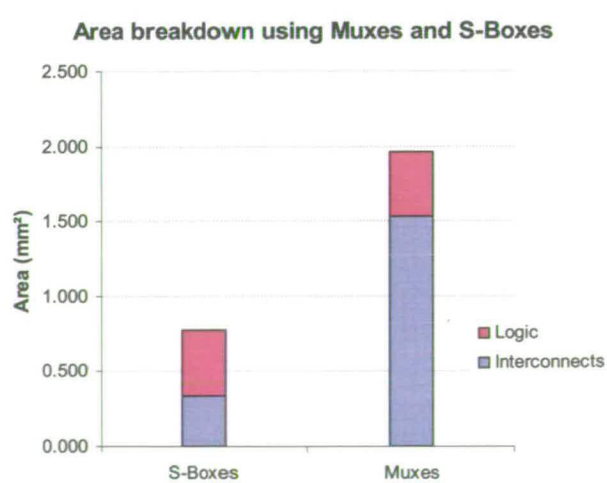


Figure 5-26: Break down of area in RICA using both multiplexers and s-boxes as interconnects

For the power consumption, the detailed measurement was only done for multiplexer-based interconnects, as no layout for an array with s-boxes was done. The breakdown is shown in Figure 5-27. On the UMC 130nm technology used the leakage power was measured to be around 10% of the total power consumption. The contribution of the interconnects to the total power consumption was found to be on average 11%. This low overhead signifies that the chosen granularity and breakdown of functional units is efficient.

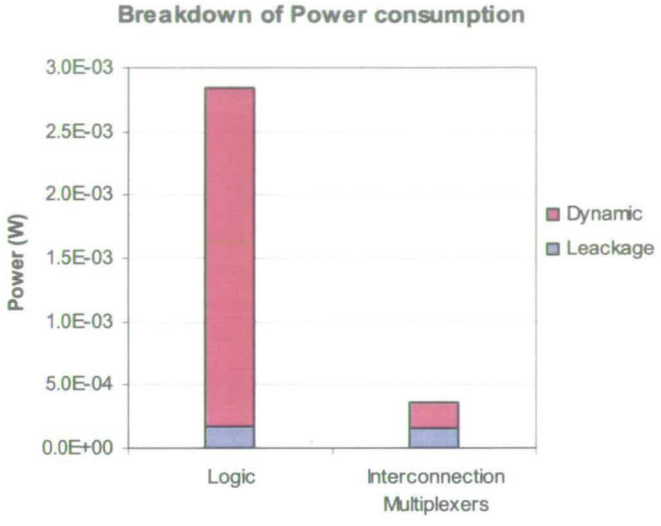


Figure 5-27: Break down of power consumption in RICA using multiplexers as interconnects.

6.6. Conclusion

The table below compares the proposed RICA architecture to DSRAs, FPGAs, DSP and VLIW technologies. The performance measured demonstrates attractive results regarding the four important requirements for future systems: low cost, low power-consumption, high flexibility and simple design-flow. When compared to current technologies, RICA outperforms low-power DSP architectures such as the TI C55x with up to a 3 times higher throughput and with 2-6 times less power consumption. It should be noted that the degree of power savings depends on the amount of control operations in the program. When compared to current VLIW processors, RICA considerably reduces the number of required clock cycles in applications containing numerous dependent instructions since it allows the execution of both dependent and independent instructions concurrently, which solves the problem of statistical ILP-limit faced by VLIW. In terms of timing performance, RICA achieves similar timing to the VLIW for datapath application, while being up to 50% slower in control intensive application. This is due to the fact that the VLIW circuitry has been handcrafted to achieve 600MHz operating frequency [113]. Nevertheless, RICA can achieve up to 6 times less power than the VLIW.

RICA vs. DSRA
▪ Programmable using a high-level C language. ▪ DSRAs allow better lower-level tuning. ▪ RICA is easier to interface with other SoC elements using the data-memory and direct-memory-access (DMA). ▪ Lower area, ▪ Less configuration bits ▪ Dynamic reconfiguration
RICA vs. DSP/RISC
▪ Distributed registers, and hence lower power than centralised register file. ▪ Distributed Data memory access. ▪ Temporary register/memory access becomes wires between cells. ▪ Lower-power ▪ Higher throughput ▪ Larger program size
RICA vs. VLIW
▪ Faster for datapath computations, similar throughput for control. ▪ Similar datapath area ▪ Much lower power consumption ▪ Performance not limited by the presence of dependent instructions, no ILP limit. ▪ Distributed registers, and hence lower power than centralised register file. ▪ Distributed Data memory access. ▪ Temporary register/memory access becomes wires between cells. ▪ Larger program size when uncompressed
RICA vs. FPGA
▪ Less flexible since coarse-grain ▪ Much lower power consumption ▪ Lower area ▪ FPGAs should be able to achieve a higher degree of parallelism since there are no area limits. ▪ Programmable using a high-level C language ▪ Dynamic reconfiguration
RICA vs. ASIC
▪ Much more flexible ▪ Higher power consumption ▪ Larger area ▪ ASICs should be able to achieve a higher degree of parallelism since there are no area limits. ▪ Programmable using a high-level C language ▪ If RICA is replacing several hardwired IPs, then its distributed memory removes the need for a shared bus to communicate data between the IPs, and hence reduces power.

The measured performance of the initial array is encouraging; however, more tuning can be done on the compiler level, such as making the scheduling occurs inside *gcc*, to greatly boost the performance. Furthermore, due to the limitations of the currently used compiler, some arithmetic operations have to be optimized manually. This is especially true for applications which software implementation is completely different from the hardware one, as seen in the next chapter.

One problem in the proposed RICA architecture lacking compression is the large program memory requirements compared to other processors. Since memory consumes much power,

this can potentially affect any power saving achieved in the datapath. However, work is currently being carried on in this area to determine an efficient compression scheme to reduce the required number of program-bits while having a fast decoding time essential for dynamic reconfiguration. This can be achieved by distributing the program memory to each cell and allowing the use local program-indices to determine the activity of the cell. The compression of the program memory is also being investigated at the same time as the interconnects structure in order to find a suitable program coding format usable in an S-Box based interconnects scheme [128].

Chapter 7:

Advanced implementations on RICA

The RICA fabric can be large enough to allow making circuits containing multiple functional elements as is the case in ASIC and FPGAs. This enables us to use design techniques and optimisations that are conventional in hardware circuit designs. Since such methodologies are uncommon in normal processors, they are not automatically identified and applied by the existing *gcc* compiler. This chapter shows two examples of such optimisations: First the use of registers to implement propagation/broadcasting schemes and second the use of multilevel pipelining for increasing throughput.

Additionally, since RICA is programmable using a high-level language and it can execute both control and datapath oriented operations, it becomes possible to rapidly run large applications on the architecture: In the second part of this chapter, an mp3 audio and H.264 video codecs (which otherwise are too time-consuming and too difficult to implement on ASICs, FPGAs or DSRAs) are used to prove this programmability of RICA.

7.1. Example of manually optimised implementations

7.1.1. FIR Filter using shift-register

The conventional method of designing an FIR filter in software is to use the data-memory to store the input and the coefficients and to go through the input array multiplying each element with a coefficient. This is shown in Table 5-8 which is taken from TI's benchmarks for the TI62x [121]. In this code for a 10-tap FIR filter, which was originally designed for the 8-way VLIW, the inner-loop can be unrolled automatically by the compiler. However, the unrolled output will be abundant of dependent instructions (the sum variable) and it would not be possible to use any of the 8 ALUs of the VLIW in parallel, hence it is very inefficient for VLIW.

Table 5-8: C code for conventional FIR in software from TI benchmarks [121]

```
void A_fir_vselp_u(const short * iPtr, const short * coefPtr, short *oPtr)
{
    for (iPtr += 10, int i=0; i < N; i++)     {
        int sum = 0;
        for (int j = 1; j <= 10; j++)                // This is a 10-tap filter
                sum += (int)(short)coefPtr[j-1] * (int)(short)iPtr[i-j];
        oPtr[i] = ((sum + iPtr[i]) >> 15);
    }
}
```

If this code is compiled for RICA, then no ILP-limit problem is faced due to dependent-instructions and it executes more efficiently than on a VLIW, however there is still more room for improvement. Most of the execution time is spent in the RMEM cell for accessing the coefficients and the input data, and the same coefficients and memory locations get read several times during the full loop. A more efficient implementation can be achieved by using a hardware-like FIR filter that uses shift-registers to store the previous input values. Since conventional DSP processors do not allow implementing shift-registers, most of the existing code has been tailored for replacing such hardware-structures with memory access. However, since the RICA fabric enables mapping circuits such as shift-register, the code can be re-written to execute faster and with less RAM access, as shown in Table 5-9. The proposed code only reads an input value once and puts it through 10-shift registers (to represent the 10-taps), and in each inner-loop only one coefficient is read and multiplied by the appropriate value. In the example shown this gives an immediate 43% speed-up.

Table 5-9: C code for FIR with reduces memory access using shift-registers, similar to hardware implementations.

```
void fir_with_sr(const short * iPtr, const short * coefPtr, short *oPtr)
{
        int i,j;
        short coef_j;
        register int   r22, r23, r24, r25,   r26, r27, r28, r29,    r30, r31;
        int sum0, sum1, sum2, sum3, sum4, sum5, sum6, sum7, sum8, sum9;

        r23= r24= r25= r26= r27= r28= r29= r30= r31= 0; // Initialise

    for (i=N-1; i >= 9; i=i-10)     {
        sum0= sum1= sum2= sum3= sum4= sum5= sum6= sum7= sum8= sum9 = 0;

        for (j = 0; j < 10; j++)          {          // 10-tap filter
                r22 = (int)iPtr[(i-j)];               // Read the input mem value
                coef_j = coefPtr[j];                  // Read the coef

                sum9 += (int)(short)coef_j * r22;     // Do the calculation
                sum8 += (int)(short)coef_j * r23;
                sum7 += (int)(short)coef_j * r24;
                sum6 += (int)(short)coef_j * r25;
                sum5 += (int)(short)coef_j * r26;
                sum4 += (int)(short)coef_j * r27;
                sum3 += (int)(short)coef_j * r28;
                sum2 += (int)(short)coef_j * r29;
                sum1 += (int)(short)coef_j * r30;
                sum0 += (int)(short)coef_j * r31;

                 // Do the shifting (it is a 10-tap filter)
                r31 = r30;                    r30 = r29;
                r29 = r28;                    r28 = r27;
                r27 = r26;                    r26 = r25;
                r25 = r24;                    r24 = r23;
                r23 = r22;
        }

        for (j = 0; j < 10; j++)              // Write the 10 outputs
                oPtr[i-j] = ((sum0 + iPtr[i-j]));
    }
}
```

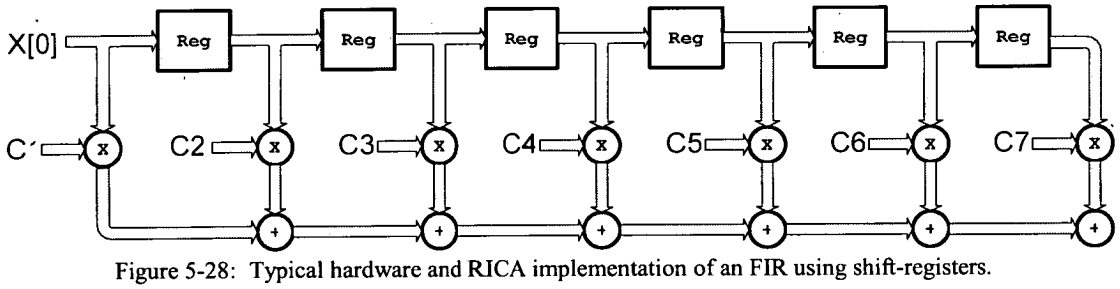


Figure 5-28: Typical hardware and RICA implementation of an FIR using shift-registers.

If we had more than 10-taps, then we would need more registers to do the shifting. In this case we either add more REG cells to the array, or we can use the 2nd bank of each REG cell. The other solution is to use the data-memory (i.e. WMEM/RMEM cells). Also, if the number of taps was fewer than 10, of if we had 12 MUL and 12 ADD we could have fitted it inside a single step and used a pipelined scheme (like the one described below for the FFT) to improve the throughput further. Staying in the same step also reduces the time need to fetch the new configuration for the next step.

Table 5-10: Measurement of improvement in shift-register based FIR filter.

	Cycles	Total time (μs)
FIR w/o SR	481	3.61
FIR using SR	339	2.04

7.1.2. Pipelined 8192-point FFT for OFDM

The 8192-point FFT (or 8k FFT) was chosen for implementation on RICA as it is a highly computational part of the Digital Video Broadcasting (DVB) standard. Here we take the example of the DVB-T standard targeting terrestrial digital broadcast; a DVB-T compliant High Definition TV (HDTV) set uses OFDM (Orthogonal Frequency Division Multiplexing) signalling to achieve the required high bandwidths[122]. As described in the DVB-T standard, the OFDM receiver uses an 8192-point FFT transform which needs to be performed within 924μs.

This FFT is usually implemented pn FPGA or ASICs, as DSP implementations are complex [123] [124]. Having this FFT implemented on a software programmable architecture would be a great advantage towards the implementation of a Software Defined Radio (SDR) on RICA.

An N-point FFT operation is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{nk}, \qquad k = 0,1,\Lambda\ N-1$$

Where the *twiddle factor* W is:

$$W_N^{nk} = e^{-j2\pi nk/N}$$

The main FFT computation requires a large number of operations, however, due to the nature of the twiddle factor W several algorithms have been designed to reduce the number of computations required; the algorithm chosen to be implemented on RICA is the Cooley-Tukey Decimation-in-Time (DIT) Radix-2 algorithm [125]; to compute the FFT for 8192 points 13 stages are required. In each stage 4096 radix-2 butterfly operations need to be carried. The input to each stage is the output of the previous stage, hence one advantage of this algorithm is that there is no need to use intermediate memory buffers for the FFT, as it can be placed on the memory location as the input.

In order to reduce the complexity of the algorithm further, the 8192-points can be divided into 6 radix-4 stages followed by one radix-2 stage. However, 13 radix-2 stages were chosen to reduce the program size and to make it easier to implement the pipeline (described below). A Radix-2 butterfly is in effect a 2-point FFT computation; it has 2 inputs x_0 and x_1 and 2 outputs y_0 and y_1, and uses the twiddle factor W_N^r :

$$y_0 = x_0 + x_1 \cdot W_N^r$$
$$y_1 = x_1 - x_1 \cdot W_N^r$$

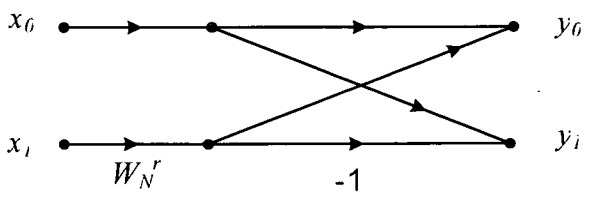


Figure 5-29: Radix-2 complex butterfly computation.

All these operations are complex operations, and hence the numbers have imaginary and real parts. A complex multiplication can be implemented using 4 real multipliers and 2 real adders. Hence, each radix-2 butterfly has 1 complex multiplication and 2 complex additions which comes down to 10 real operations.

In order to speed-up the execution of the FFT, it was decided to add the complex arithmetic functionality to the MUL and ADD cells themselves and not to do it in software. With this approach, the 32-bit ADD cell can also perform a 16-bit complex addition. The 16-bit real and imaginary part of each complex number would be combined into the 32 bits used to represent real numbers. This gives the FFT a 16-bit precision which is enough for OFDM applications, as typical FFTs for DVB-T use 12-bit processing.

The Decimation-in-Time (DIT) FFT algorithm also requires a bit-reversing operation to be performed on the 8k input either before or after the 13 stages of radix-2. The bit-reversing ability simply converts input data such as `0001101...0` to `0...1011000`, and is used to modify the addressed of the 8129 input samples. This generic bit-reversing ability has been added to the LOGIC cell, as it would be very time costly to implement it in software. With this approach the extra LOGIC cell would be used after the address-generator in the first stage when the input data is accessed for the first time.

The address-generator needed to read and write between two stages has to follow the addressing needed for the decimation-in-time algorithm. This is shown as an example for the 8-point FFT in Figure 5-30. The details for this addressing can be found in the code in Table 5-11, where the address is calculated using the variables `point` and `stride`.

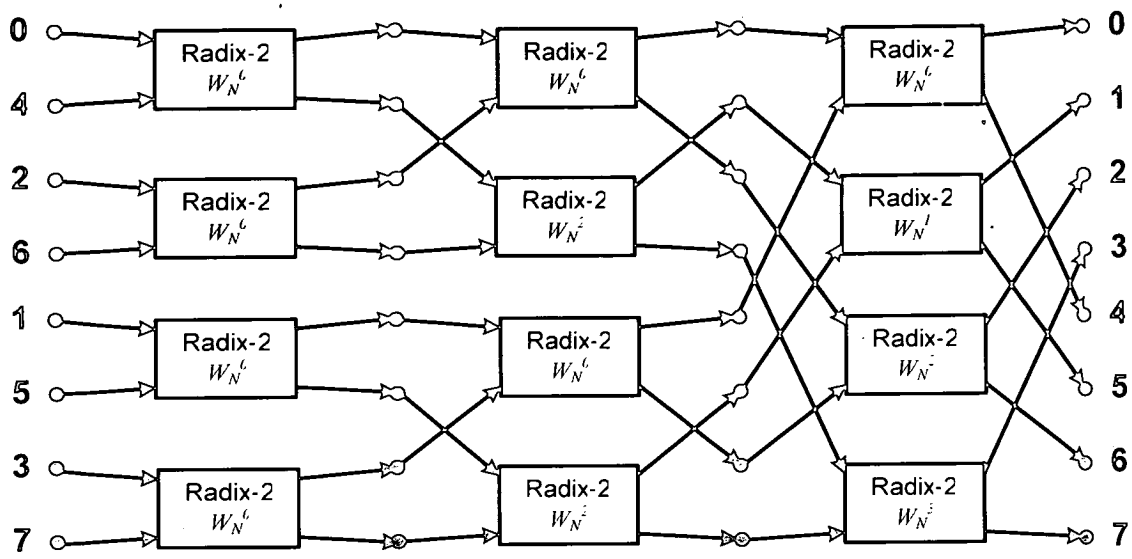


Figure 5-30: 8-point FFT computation using Radix-2 butterfly

Table 5-11: 8k FFT computation with the main loop fitting into a single step

```
for(stride=1; stride != n_points; stride *=2)  //For 8192, 13 stages of Radix-2
{
    point = 0;
    counter = 0;

    do   {
            twiddle = twiddle_table[counter];

            in0 = data[point];
            in1 = data[point+stride];

            CPLX_MUL(temp_val, in1, twiddle);
            CPLX_ADD(temp0, in0, temp_val);
            CPLX_SUB(temp1, in1, temp_val);

            data[point] = temp0;
            data[point+stride] = temp1;

            temp_mux1 = point + stride*2;
            temp_mux2 = temp_mux1 - n_points + 1;
            point = (temp_mux1 >= n_points) ?  temp_mux2 : temp_mux1;

            counter++;
      } while(counter < half_n);
}
```

If the code of Table 5-11 is compiled, then we can fit the main loop calculation into a single step if we have the following resources in the array: 8x ADD, 2x MUL and 2x SHIFT. This is shown in Figure 5-31. As it can be seen, the longest-path delay in this step would the path RMEM-MUL-ADD-WMEM, which is around 27 ns. Since this loop is executed 13·4096 = 53,248 times, it would takes 1437μs to finish the 8k FFT calculation. As described earlier, this is too long for the DVB-T standard.

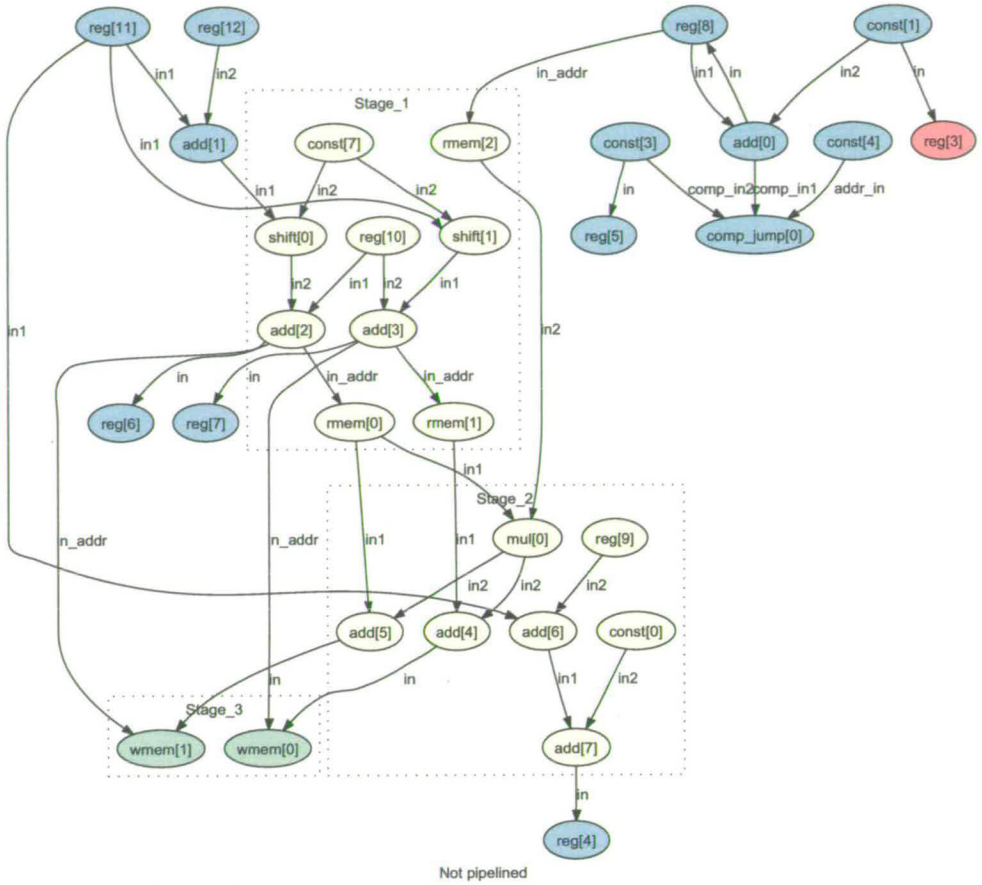


Figure 5-31: Main loop step if compiled from code (counter not shown)

To improve the throughput we can employ a 3 stage pipe: between the RMEM and the arithmetic operation, and between the arithmetic operation and the WMEM. In this case, the longest path becomes 10 ns, which reduces the time needed to compute the whole FFT to 530μs, making the implementation compatible with DVB-T. To make this work we would need to execute 2 extra cycles to fill the pipeline and 2 cycles to empty it.

The fact that the whole loops fits into a single *step* and that this *step* loops back to itself allows achieving this high performance; in this case the configuration for the array does not change and there is no need to fetch a new instruction from the program memory. This gives near ASIC-like speed since the only overhead compared to ASICs are the relatively light interconnects between the cells.

Table 5-12: Comparison of the performance of FFT with and without pipeline.

	Cycles	Longest-path (ns)	Total time (μs)
FFT w/o pipeline	53248	27	1437
FFT w pipeline	53248 + 4	10	530

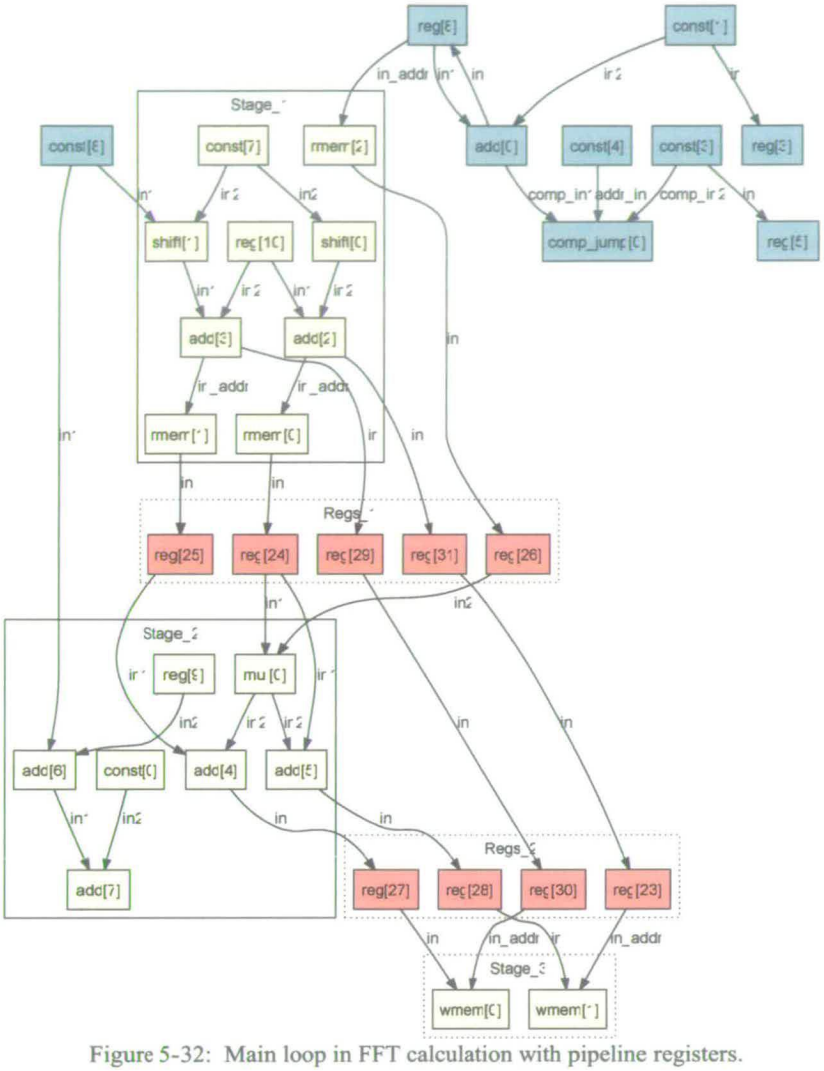


Figure 5-32: Main loop in FFT calculation with pipeline registers.

When running the pipelined FFT the datapath exhibits an energy consumption of 5.2 mW. The same 8k FFT would required around 200,000 cycles to run on a TIC62x VLIW – hence an operating frequency of 377MHz would be required to complete the calculation in 530µs, which means that the datapath would consume 192mW, as it is characterised at 0.51 mW/MHz. This means that RICA's datapath is around 37 time more energy efficient that the VLIW.

For the purpose of this experiment, this modification and the addition of the pipeline registers was manual. However, in the future the scheduler should identify loops that fit into single steps and should try to add the pipeline automatically.

7.2. Larger systems: MP3 Audio and H264 Video

Large systems that are impractical to design using HDL such as multimedia applications like mp3 audio and H.264 video decoding; these applications contain large amounts of conditional execution and operations that make it a requirement to use a high-level description language to program and maintain the code as well as to reduce the design cycle since these standards

keep changing. To demonstrate the programmability of RICA, the open-source mp3 decoder *libmad* [126] and the open-source H.264 decoding module from *ffmpeg* [127] were compiled and profiled. The untouched code was compiled with no modifications to the actual audio/video decoding elements were done – only some output printing functions were disabled as they are not support on the RICA hardware. The performance values shown are for the same 64-cell sample RICA described in the previous chapter. The same code was also compiled for ARM9TDMI, which is a processor specially tailored for multimedia applications. The speed and energy consumptions of the solutions are shown in the tables below; the values shown for ARM9TDMI assume that it is running at its maximum frequency of 250MHz and that it consumes 0.25 mW/MHz [116] (cache is disabled and ideal situation is supposed), while the ARM7TDMI-S runs at its maximum frequency of 110MHz.

For the mp3 benchmark, a two-frames long stereo 64kbps sample input was used. The time and energy consumption shown are the ones measured for the duration of recoding the 2 frames. The results (Table 5-13) show that RICA decodes the frames 3.4 times faster than ARM9 with a datapath energy consumption 10.8 times lower.

Equally attractive results are measured for decoding H.264 frames (Table 5-13 and Figure 5-33) where RICA is 13.8 times faster than ARM7 and having 6.7 times less energy. The sample video used contains two QCIF (177x144) frames at 460 kbps data rate.

Table 5-13: Performance comparison of the libmad mp3 decoder on RICA and ARM9 (2 frames)

	ARM9TDMI (250MHz)	RICA
Datapath energy Consumption (uJ)	127.60	11.80
Decoding speed (ms)	2.06	0.60

Table 5-14: Performance comparison of the ffmpeg H264 decoder on RICA, ARM9 and ARM7 (2 QCIF frames)

	ARM9TDMI (250MHz)	ARM7TDMI (110 MHZ)	RICA
Energy Consumption (mJ)	2.15	0.74	0.11
Decoding speed (ms)	39.60	111	8.03

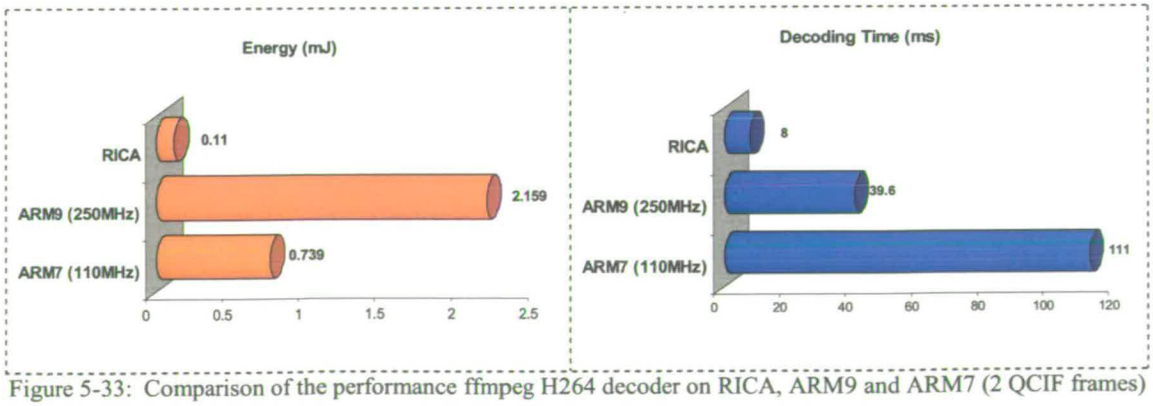


Figure 5-33: Comparison of the performance ffmpeg H264 decoder on RICA, ARM9 and ARM7 (2 QCIF frames)

The performance shown is for directly-compiled and unoptimised code. Important speed-ups (around 2-4 times) should be achievable using similar techniques to the ones described earlier such as shift-registers and pipelining, which would make RICA to easily support future H.264 decoding of large frames (e.g. D1 720x480) at real time – such an implementation is impossible today using a programmable solution that is usable in portable applications. The list in Table 5-15 shows the percentage of time spent in each function while decoding two different video sequences of 20 D1 frames (one with CABAC coding and the other with CAVLC). Such a profiling gives an idea of which functions have a priority in being optimised and optionally hand-coded to increase the performance. In this case these functions would the filtering ones (`_decode_residual`, `_decode_cabac_residual`, `_filter_mb`, `_h264_?_loop_filter_luma_c`, `_put_h264_qpel8_?_lowpas`). It can also be seen that the initialisation function `_memset` occupies quite a large percentage – this is only the case because the hardware has only decoded 20 frames and running the decoder for a longer time would reduce the relative percentage of this function. Nevertheless, the `_memset` function used operates on a byte level. Since RICA has multiple memory banks that can allow simultaneous memory writing, a direct 4 times speed-up can be achieved by rewriting `_memset` to simultaneously write 4 bytes.

Table 5-15: Profiling of the ffmpeg H264 decoder on RICA, running through 20 D1 frames

D1 720x480, 20 Frames, CAVLC, 13.6 fps	
_memset	14.17%
_put_h264_qpel8_h_lowpass	13.58%
_put_h264_qpel8_v_lowpass	10.82%
_decode_residual	9.22%
_put_h264_qpel8_hv_lowpass	7.05%
_ff_h264_idct8_add_c	6.36%
_ff_h264_idct_add_c	5.92%
_put_h264_chroma_mc8_c	5.49%
_decode_mb_cavlc	4.85%
_memcpy	4.78%
_hl_decode_mb	3.20%
_fill_caches	2.28%

D1 720x480, 20 Frames, CABAC, 19.6 fps	
_decode_cabac_residual	10.52%
_filter_mb	10.08%
_memset	8.88%
_put_h264_qpel8_h_lowpass	7.92%
_h264_v_loop_filter_luma_c	6.06%
_put_h264_qpel8_v_lowpass	5.99%
_put_h264_qpel8_hv_lowpass	5.55%
_h264_h_loop_filter_luma_c	5.44%
_ff_h264_idct_add_c	5.27%
_decode_mb_cabac	4.14%
_put_h264_chroma_mc8_c	3.56%
_hl_decode_mb	3.01%

7.3. Conclusion

Due to the limitations of the compiler some arithmetic operations have to be optimised manually. This is especially true for applications for which the software implementation is completely different from the hardware one, e.g. FIR, where in hardware we would naturally use shift-registers, while existing software implementations use memory copying and access. This use of shift-registers was demonstrated for an FIR filter and showed a 43% speed up on RICA as memory access got reduced. The modification was done on a C-language level.

The second hardware-like technique tested is programmable pipelines. The example used is a compute intensive 8k FFT calculation. Pipeline-optimisation was performed on a single step

level by manually changing the scheduled code to add registers between the instruction cells or long paths. This resulted in 2.5-3 times throughput increase over the non-pipelined version. These useful hardware design techniques can be easily added in the future to the compiler to make their usage automatic, and hence there would be no need for manual low-level coding. Furthermore, since RICA is programmable in C, it was possible to compile large and complex systems to demonstrate its programmability feature. An open-source MP3 audio decoder and H.264 video decoder were directly compiled in a straightforward way in a week time. The measured performance and power consumption on RICA compares favourably to other solutions: RICA is around 10x faster and more power efficient than ARM9. However, as with any CPU processor, there is more room for improvements by manually writing critical operations in assembly/netlist level. Future versions of the compiler and scheduler should help making this type of optimisations more automatic.

Chapter 8:

Conclusion

In this work, the initial approach to develop a solution for the flexibility problem in System-on-Chip architectures was to focus on coarse-grain Domain-Specific Reconfigurable Arrays (DSRAs) as a mean to provide a solution with high throughput and low power-consumption when compared to other alternatives such as embedded FPGAs and DSP processors. To make any domain-specific scheme usable for a large number of applications, a framework for creating such arrays was designed. The generated DSRAs have an FPGA-like structure as this provides a reasonable uniformity and allows the reuse of existing software. From a structure point of view, the DSRAs differ from FPGAs in that they are coarse grain heterogeneous arrays.

Two sample DSRAs were generated for video coding applications; the measured performance indicates that DSRAs can indeed be classified as a compromise between FPGAs and ASICs in terms of flexibility, power, area and delays. It was also found that the performance of a DSRA can be optimised further by tailoring the directivity and the circuit design of interconnects; this gives improvements in power and area at the cost of increased delays and lower routability.

To generate a DSRA, the designer has to manually identify the algorithms targeted and the operations needed in order to create the clusters for the array. In the future, once several applications have been designed using DSRAs, a library of clusters can be created to reduce this lengthy DSRA design-time. In short, the rapid deployment of DSRAs depends on the existence of such a library. Another limitation to DSRAs is the fact that in the same way as ASICs and FPGAs, they have to be programmed at low-level using a time-consuming Hardware Description Language.

DSRA vs. FPGA	DSRA vs. ASIC
▫ Lower area ▫ Much lower power consumption ▫ Higher frequency ▫ Less flexibility	▫ Much higher flexibility ▫ Higher power consumption ▫ Higher area ▫ More delays

To overcome these problems, the second proposed approach was to use an architecture called the Reconfigurable Instruction Cell Array (RICA). By using so called *instruction-cells* that accept processor-like instructions, it becomes possible to map a compiled software representation of an algorithm directly to the reconfigurable fabric. Coupled with the ability to dynamically and rapidly reconfigure the array, running complete software programs on RICA is feasible. The open-source *gcc* C compiler was modified to compile software to RICA. Several C benchmark algorithms were tested, and RICA demonstrated attractive results compared to other architectures. RICA outperformed current low-power DSP architectures such as the TI C55x by providing up to a 3 times higher throughputs and with 2-6 times less power consumption in the data-path. When compared to current high-end VLIW processors RICA achieves similar timing for datapath applications, while being up to 50% slower in control intensive applications. This is due to the fact that the VLIW circuitry has been handcrafted to achieve high operating frequencies. Nevertheless, RICA achieved up to 6 times less power than the VLIW using standard code, and up to 37 times less in the case of the pipelined FFT.

The straightforward programmability of RICA made it also possible to run existing large systems such as an *mp3* audio decoder and an H.264 video decoder only after a few days design-time. It was also demonstrated that by manually programming RICA at low-level it

becomes possible to use hardware-like optimisations that are not usually found in processors, mainly due to the limitations of the used compiler. This included the use of elements such as multiplexers, shift-registers and pipeline registers to increase throughput and reduce memory access.

RICA vs. DSRA	RICA vs. DSP/RISC
▪ Programmable using a high-level C language ▪ DSRAs allow better lower-level tuning ▪ RICA is easier to interface with other SoC elements using the data-memory and direct-memory-access (DMA) ▪ Lower area ▪ Less configuration bits ▪ Dynamic reconfiguration	▪ Distributed registers, and hence lower power than centralised register file ▪ Distributed Data memory access ▪ Temporary register/memory access becomes wires between cells ▪ Lower-power ▪ Higher throughput ▪ Larger program size

RICA vs VLIW	RICA vs FPGA
▪ Faster for datapath computations, similar throughput for control ▪ Similar datapath area ▪ Much lower power consumption ▪ Performance not limited by the presence of dependent instructions, no ILP limit ▪ Distributed registers, and hence lower power than centralised register file ▪ Distributed Data memory access ▪ Temporary register/memory access becomes wires between cells ▪ Larger program size	▪ Less flexible since coarse-grain ▪ Much lower power consumption ▪ Lower area ▪ FPGAs should be able to achieve a higher degree of parallelism since there are no area limits. ▪ Programmable using a high-level C language ▪ Dynamic reconfiguration

RICA vs. ASIC
▪ Much more flexible ▪ Higher power consumption ▪ Larger area ▪ ASICs should be able to achieve a higher degree of parallelism due to reduced area limits ▪ Programmable using a high-level C language ▪ If RICA is replacing several hardwired IPs, then its distributed memory removes the need for a shared bus to communicate data between the IPs, and hence reduces power

Future work in the RICA domain would need to mainly focus on two aspects: First, the improvements of the software-tool flow to optimise further the design. This includes using improved instruction scheduling algorithms, integrating the scheduling as part of the compilation phase and allowing the compiler to identify hardware-like optimisations that are possible on RICA. The second aspect would concentrate on the hardware design of the interconnects to allow a better scalability of the array (i.e. allow the usage of 500+ cells) along with the design of methods for reducing the program memory usage, as this has considerable part of the total power and area consumption on the chip. Several program-memory compression schemes are possible, including the use of distributed memories and

local program-counters to remove redundant data, as well as the use of path-encoding methods [128].

In the future, the current architecture can be heavily optimised by adding asynchronous logic capabilities to the Instruction-Cells. Completion-detection signals can be created at the output of each Instruction-Cells to signal when the next cell in sequence should start operation. This would completely eliminate any need for the CLK_DIV cell as each step would only take the exact time it needs to finish the calculation. This helps in further reducing the program size as no configuration data is needed for CLK_DIV.

In terms of silicon utilisation, adding multithreading capabilities to the architecture would dramatically increase it along with increasses in the degree of parallel operations that can be executed. Having multiple JUMP cells and multiple program-counters coupled with the ability to dynamic schedule the silicon resources between multiple tasks would create an ideal system architecture with a very high degree of scalability, flexibility and an extremely high performance per silicon area, unachievable in any other architecture.

Achievements

Domain-specific reconfigurable arrays

- Hardware design of DSRA programmable fabric
- Framework and tools to generate arrays according to defined clusters
- Tools to program (including routing) and test the arrays at various stages of the SoC design-flow
- Library of interconnect structures that can be used to tailor the arrays towards the application
- Optimised clusters useful in video coding and filtering applications
- Hardware design of two arrays targeting MPEG video decompression

Reconfigurable Instruction Cell Array architecture

- Hardware design of RICA system composed of heterogeneous instruction-cells, programmable interconnects and memory interfaces
- Tool to generate RICA arrays with customisable numbers and functionalities of Instruction Cells
- Modified *gcc* compiler for generating RICA-specific assembly
- Scheduling tool to extract instruction parallelism from assembly
- Optimised software implementations of DSP operations on RICA

Contribution to knowledge

This study was aimed at providing a deeper understanding of practices for achieving optimised SoC design in terms performance and costs. Tackling this issue from the point-of-view of flexibility and the generality provided by hardware verified the existing conception that the more specific the hardware, the higher the costs and the higher the performance are, and vice-versa. This study showed that in order to create realistic designs at a domain-specific level – a hybrid level between the extreme general FPGAs and the extreme hardwired solutions ASICs – another general layer is required, which consists of a software-framework to generate these domain-specific hardware designs.

The presented work concentrated also on finding middle-grounds between existing extremes of reconfigurable architectures from the point-of-view of reconfiguration time; i.e. somewhere between the extremely infrequent FPGA reconfiguration and the single-cycle reconfiguration in DSPs. It was proven that efficient silicon architectures can be achieved by combining a reconfigurable fabric interconnected in an FPGA-style along with an atomic granularity similar to ALUs in DSPs and coupled to an instruction execution and control mechanism similar to processors. This resulting architecture can execute both control and datapath intensive code at performances currently separately obtainable using DSPs (for control) and FPGA (for datapath).

Furthermore, with this approach the hardware-design flow stays at high-level C-language. It can be seen as if the hardware design methodology becomes a mix between C and HDL: Big functional loops can be conceptually thought of as HDL (being described in C), while the program flow and control operations are done in the easy and conventional way in C. This solves an enormous problem faced today in terms of finding new ways to program parallel systems.

A. Sample RICA cells with instruction set

The supported Cells/Instructions are shown in the following table:

Cell	Supported Configurations	Inputs	Outputs
ADD	{ADD, SUB}+{SI, HI, QI}	2: A, B	1: O
COMP	{EQ, NE, GTS, GES, LTS, LES, GTU, GEU, LTU, LEU} + {SI, HI, QI}	2: A, B	0
CONST	{#Num}	0	1: O
DIV	{DIV_SIG, DIV_UNSIG} + {SI, HI, QI}	2: A, B	1: O
JUMP	{IF_T, IF_F, ALLWAYS}	1: ADDR	1: NL
LOGIC	{SE, ZE, AND, OR, XOR, NOT, NEG} + {SI, HI, QI}	2: A, B	1: O
MUL	{MUL_SIG, MUL_UNSIG} + {SI, HI, QI}	2: A, B	1: O
REG	{WRITE,READ}	1: I	1: O
RMEM	{NO_OFF} + {SE, ZE} + {SI, HI, QI}	2: ADDR, OFFSET	1: DATA
SHIFT	{SLL, SRA, SRL} + {SI, HI, QI}	2: A, B	1: O
WMEM	{Enable} + {NO_OFF}{SI, HI, QI}	3: ADDR, DATA, OFFSET	0

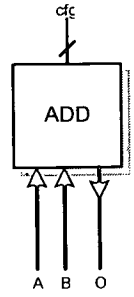
These are the same operations supported on the OR32 implementation of the OpenRISC, hence anything that compiles and runs on the OR32 can be converted to this architecture. This is similar to the instruction set provided in the ARM7.

The `SI`, `HI`, `QI` option specify the width of the data operated on:

```
SI : Single Integer = 32-bits
HI : Half Integer = 16-bits
QI : Quarter Integer = 8-bits
```

8.1. ADD

Configuration bits: 3 bits	
C0	**C1-C2**
0: Addition 1: Subtraction	00: SI 01: HI 10: QI



I/O Pin	Dir	Size	Description
A	In	32-Bit	Input 1 operand
B	In	32-Bit	Input 2 operand
O	Out	32-Bit	Result of Add/sub operation

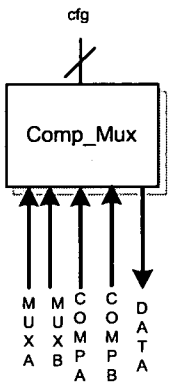
Simplified operation:

```
O = C0 ? A-B : A+B
```

Comments:

- Standard Addition and Subtraction
- Combinatorial cell

8.2. COMP_MUX



Number of configuration bits: 6

Configuration bits: 6 bits	
C0-C3	**C4-C5**
0000: EQ (Equal) 0001: GTS (Greater Than - Signed) 0010: GES (Greater than or Equal to - Signed) 0011: GTU (Greater Than - Unsigned) 0100: GEU (Greater than or Equal to - Unsigned) 1000: ZERO (Compare to Zero) 1001: GTZS (Compare to Zero) 1010: GEZS (Compare to Zero) 1011: GTZU (Compare to Zero) 1100: GEZU (Compare to Zero)	00: SI 01: HI 10: QI

I/O Pin	Dir	Size	Description
MUX_A	In	32-Bit	Multiplexer Input 1
MUX_B	In	32-Bit	Multiplexer Input 2
COMP_A	In	32-Bit	Comparator input 1
COMP_B	In	32-Bit	Comparator input 2
DATA_OUT	Out	32-Bit	Multiplexer Output

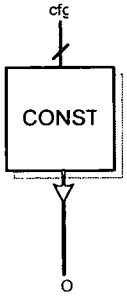
Simplified operation:

```
DATA_OUT = (COMP_A § COMP_B) ? MUX_A : MUX_B
```

Comments:

- MUX_B is set to Zero when compare to Zero selected
- Combinatorial cell

8.3. CONST



Configuration bits: 32 bits
C0-C31
The required 32-bit output constant

I/O Pin	Dir	Size	Description
O	Out	32-Bit	Output constant

Simplified operation:

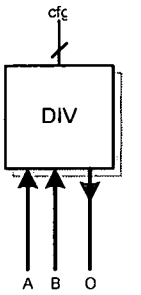
```
O = Constant
```

Comments:

- Provides constant value through configuration program memory
- Combinatorial cell

8.4. DIV

Number of configuration bits: 3



Configuration bits: 3 bits	
C0	**C1-C2**
0: Singed Division 1: Unsigned Division	00: SI 01: HI 10: QI

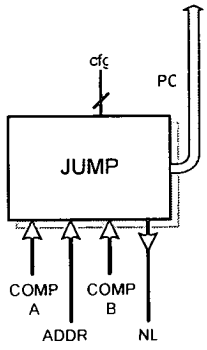
I/O Pin	Dir	Size	Description
A	In	32-Bit	Input 1 operand
B	In	32-Bit	Input 2 operand
O	Out	32-Bit	Result of division

Simplified Operation:

```
O = A / B
```

8.5. COMP_JUMP

Number of configuration bits: 9



Configuration bits: 3 bits			
C0-C1	**C2**	**C3-C6**	**C7-C8**
00: GO TO NEXT STEP 01: JUMP ALWAYS 10: JUMP IF FLAG IS HIGH 11: JUMP IF COND IS LOW	0: Relative Address 1: Absolute Address	0000: comp_Eq 0001: comp_GTS 0010: comp_GES 0011: comp_GTU 0100: comp_GEU 1000: comp_ZERO 1001: comp_GTZS 1010: comp_GEZS 1011: comp_GTZU 1100: comp_GEZU	00: SI 01: HI 10: QI

I/O Pin	Dir	Size	Description
ADDR	In	32-Bit	Input Address
COMP_A	In	32-Bit	Comp In
COMP_B	In	32-Bit	Comp In
NL	Out	32-Bit	Address of Next Location

Operation:

C3-C6 performs a comparison operation on COMP_A and COMP_B

C2 indicates if the address is in absolute or relative mode

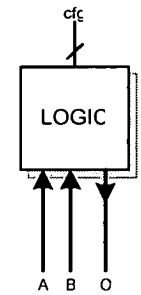
C0-C1 bits decide what sort of jump operation to perform. The flag is given from the output of COMP-A and COMP-B

Comments:

- NL output is the address that would occur if the jump is not executed. This would be the return address from a function; usually stored in the Link Register.
- The PC output goes into a decoder and then it gets converted into an address for the Program RAM.
- When nothing is connected to the cell, it acts as an instruction controller and keeps incrementing the program counter (i.e. GO TO NEXT STEP)
- Cell clocked by the CLK_DIV

8.6. LOGIC

Number of configuration bits: 6



Configuration bits: 6 bits	
C0-C3	**C4-C5**
0000: SE (Sign Extend) 0001: ZE (Zero Extend) 0010: AND (Bitwise AND operation) 0011: OR (Bitwise OR operation) 0100: XOR (Bitwise XOR operation) 0101: NOT (Bitwise Inverse operation) 0110: NEG (2 Complement negation)	00: SI 01: HI 10: QI

I/O Pin	Dir	Size	Description
A	In	32-Bit	Input 1 operand
B	In	32-Bit	Input 2 operand
O	Out	32-Bit	Result of operation

Simplified operation:

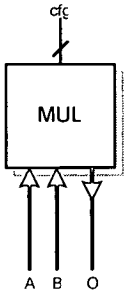
```
O =   A (Bitwise operation) B
```

Comments:

- Bitwise logic operations.
- Combinatorial cell

8.7. MUL

Number of configuration bits: 3



Configuration bits: 3 bits	
C0	**C1-C2**
0: Signed Multiplication 1: Unsigned Multiplication	00: SI 01: HI 10: QI

I/O Pin	Dir	Size	Description
A	In	32-Bit	Input 1 operand
B	In	32-Bit	Input 2 operand
O	Out	32-Bit	Result of multiplication

Simplified operation:

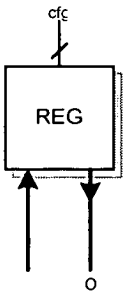
```
O =  A x B
```

Comments:

- Signed and unsigned multiplication
- Combinatorial cell

8.8. REG

Number of configuration bits: 3



Configuration bits: 3 bits		
C0	**C1**	**C2**
0: Write Bank 1 1: Write Bank 2	0: Read Only 1: Write on the next positive clock edge	0: Read Bank 1 1: Read Bank 2

I/O Pin	Dir	Size	Description
I	In	32-Bit	Data Input to write
O	Out	32-Bit	Output of register content

Simplified operation:

Read

```
O =  Reg_Bank[C2]
```

Write

```
Reg_Bank[C0] = I
```

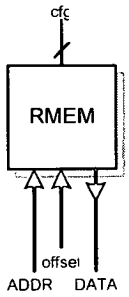
Comments:

- Cell clocked by the CLK_DIV
- Each cell contains 2 32-bit registers, bank 1 and bank2. Only one of these bank is accessible for reading or writing at any particular step. The possible combinations achievable are:

```
READ_B1
READ_B2
READ_B1_WRITE_B1
READ_B1_WRITE_B2
READ_B2_WRITE_B1
READ_B2_WRITE_B2
```

8.9. RMEM

Number of configuration bits: 4



Configuration bits: 4 bits		
C0	**C1**	**C2-C3**
0: Use Zero Offset 1: Use OFFSET	0: Zero Extend 1: Sign Extend	00: SI 01: HI 10: QI

I/O Pin	Dir	Size	Description
ADDR	In	32-Bit	Address input
DATA	Out	32-Bit	Data from memory
OFFSET	In	32-Bit	Offset

Simplified operation:

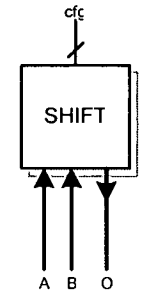
```
DATA =  DATA_RAM [ ADDR + OFFSET]
```

Comments:

- Read interface to the Data RAM banks.
- Cell clocked by the CLK_DIV

8.10. SHIFT

Number of configuration bits: 4



Configuration bits: 4 bits	
C0-C1	**C2-C3**
00: SLL (Shift Left Logical) 01: SRA (Shift Right Arithmetic) 10: SRL (Shift Right Logical)	00: SI 01: HI 10: QI

I/O Pin	Dir	Size	Description
A	In	32-Bit	Input 1 operand
B	In	32-Bit	Input 2 operand
O	Out	32-Bit	Result of shifting

Simplified operation:

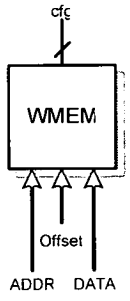
```
O =  [C0-C1] ?  A >> (B % 32) : A << (B % 32)
```

Comments:

- Logical Shift Left, Logical Shift Right and Arithmetic Right Shift supported
- Combinatorial cell

8.11. WMEM

Number of configuration bits: 4



Configuration bits: 4 bits		
C0	**C1**	**C2-C3**
0: Write Disable 1: Write Enable	0: Use no Offset 1: Use Offset	00: SI 01: HI 10: QI

I/O Pin	Dir	Size	Description
ADDR	In	32-Bit	Address input
DATA	In	32-Bit	Data from memory
OFFSET	In	32-Bit	Offset

Simplified operation:

```
If(C0 == 1) RAM[ ADDR + OFFSET] = DATA
```

Comments:

- Write interface to the Data RAM banks
- Cell clocked by RRC

B. Publications arising from this work

Publications from this work

Under Review

- S. Khawam, T. Arslan, "Frame for the design and implementation of Domain Specific Reconfigurable Arrays", *Submitted to IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, April 2006

- S. Khawam, I. Nousias, M. Milward. Y. Ying, T. Arslan; "The Reconfigurable Instruction Cell Array", *Submitted to IEEE Transactions on VLSI Systems Special Section on Configurable Computing Design*, May 2006

Published

- S. Khawam, I. Nousias, M. Milward. Y. Ying, T. Arslan, "Reconfigurable Instruction Cell Array", *UK Patent Office*, UK Patent Application Number 0508589.9, April 2005

- S. Khawam, S. Baloch, A. Pai, I. Ahmed; N. Aydin; T. Arslan; F. Westall; "Efficient Implementations of Mobile Video Computations on Domain-Specific Reconfigurable Arrays", *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2004. Proceedings Volume 2, 16-20 Feb. 2004

- S. Khawam, T. Arslan, F. Westall; "Embedded reconfigurable array targeting motion estimation applications" *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on* Volume 2, 25-28 May 2003 Page(s):II-760 - II-763 vol.2

- S. Khawam, T. Arslan; "Switch-box design for synthesizable coarse-grain arrays for system-on-chip applications", *Field-Programmable Technology (FPT), 2004. Proceedings. 2004 IEEE International Conference on*, 2004 Page(s):465 – 468

- S. Khawam, T. Arslan, F. Westall; "Domain-specific reconfigurable array for Distributed Arithmetic", *13th International Conference on Field Programmable Logic and Applications (FPL)* 2003

- S. Khawam, T. Arslan, F. Westall; "Synthesizable reconfigurable array targeting distributed arithmetic for system-on-chip applications", *Parallel and Distributed Processing Symposium (PDPS / RAW), 2004. Proceedings. 18th International* 26-30 April 2004 Page(s):150

- S. Khawam, T. Arslan, F. Westall; "Unidirectional switch-boxes for synthesizable reconfigurable arrays", *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on* 20-23 April 2004 Page(s):293 – 295

Publications influenced by this work

- Y. Ying, I. Nousias, M. Milward. S. Khawam, T. Arslan; "System-level Scheduling on Instruction Cell Based Reconfigurable Systems", *Automation and Test in Europe Conference and Exhibition (DATE), 2006.* Proceedings Volume 3, 6-10 March 2006

- Cheng Zhan; T. Arslan, S. Khawam, I. Lindsay; "A domain specific reconfigurable Viterbi fabric for system-on-chip applications", *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, Volume 2, 18-21 Jan. 2005 Page(s):916 - 919 Vol. 2

- Zhenyu Liu; T. Arslan, S. Khawam, I. Lindsay; "A high performance synthesisable unsymmetrical reconfigurable fabric for heterogeneous finite state machines", *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, Volume 1, 18-21 Jan. 2005 Page(s):639 - 644 Vol. 1

- A. Olugbon, S. Khawam, T. Arslan, I. Nousias, I. Lindsay; "An AMBA AHB-based reconfigurable SoC architecture using multiplicity of dedicated flyby DMA blocks", *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific* Volume 2, 18-21 Jan. 2005 Page(s):1256 - 1259 Vol. 2

- K. Katsoulakis, T. Arslan, T. Kirkham; Khawam S.; "A Low-Power Reconfigurable Datapath for Advanced Speech Coding Algorithms", *Parallel and Distributed Processing Symposium, (PDPS / RAW) 2005. Proceedings.* 19th IEEE International 04-08 April 2005 Page(s):147b - 147b

- L. Zhenyu, S. Khawam, T. Arslan, A. Erdogan,; "A Low Power Heterogeneous Reconfigurable Architecture For Embedded Generic Finite State Machines"; *Proceedings of SOC Conference, 2005. IEEE International* 25-28 Sept. 2005 Page(s):113 – 114

- Z. Cheng, S. Khawam, T. Arslan, I. Lindsay; "Architecture and design methodology for synthesizable reconfigurable array targeting wireless system-on-chip applications",

Proceedings of SOC Conference, 2005.. IEEE International, 25-28 Sept. 2005 Page(s):93 - 94

- Z. Cheng, S. Khawam, T. Arslan, I. Lindsay; "Efficient implementation of trace-back unit in a reconfigurable Viterbi decoder fabric"; *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on* 23-26 May 2005 Page(s):1048 - 1050 Vol. 2

- Z. Cheng, S. Khawam, T. Arslan; "Domain specific reconfigurable fabric targeting Viterbi algorithm", *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, 2004 Page(s):363 – 366

- I. Ahmed, T. Arslan, S. Khawam; "Video transmission through domain specific reconfigurable architectures over short distance wireless medium utilizing Bluetooth IEEE 802.15.1 standard", *SOC Conference, 2004. Proceedings. IEEE International*, 12-15 Sept. 2004 Page(s):7 - 10

C. References

[1] "Virtex-4 User Guide 1.5", *Xilinx*, San Jose, 2006

[2] "Stratix-II", *Altera*, Altera San Jose, 2005

[3] V. George, H. Zhang J. Rabaey; "The design of low energy FPGA", *Proceedings. 1999 International Symposium on Low Power Electronics and Design*, pp. 188-193. 1999

[4] V. George, "Low Energy Field-Programmable Gate Array", *PhD Thesis, Univeristy of California*, Berkeley . 2000

[5] I. Bryant, Y. Tanurhan, "The Actel Embeddable FPGA Core", Actel Corporation, 2001

[6] N. Kafafi, K. Bozman, S.J.E. Wilton, "Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores", *om3*

[7] E. Tau, D. Chen, I. Eslick, J. Brown, and A. DeHon, "A First Generation DPGA Implementation," *FPD'95, Canadian Workshop of Field-Programmable Devices*, May 1995.

[8] A. Marshall, J. Vuillemin, B. Hutchings; "A Reconfigurable Arithmetic Array for Multimedia Applications"; *ACM International Symposium on FPGA*, Monterey, CA, Feb 1999

[9] K. Leijten-Nowak, A. Katoch; "Architecture and implementation of an embedded reconfigurable logic core in CMOS 0.13μm", *ASIC/SOC Conference, 2002. 15th Annual IEEE International* , pp. 3 -7

[10] "D-Fabrix array", Elixent Ltd, Bristol, 2003

[11] T. Arslan, H. I. Eskikurt, D.H. Horrocks; "Configurable Structures for a primitive operator digital filter FPGA. IEEE Workshop Signal Processing Systems" *SIPS-97* . 1997

[12] B. Hounsell, T. Arslan, "Programmable multiplierless digital filter array for embedded SoC applications", *IEE Electronics Letters*. 2001

[13] J. Hammes, B. Rinker, W. Bohm, W. Najjar, B. Draper, R. Beveridge, "Cameron: high level language compilation for reconfigurable systems,; *Parallel Architectures and Compilation Techniques", 1999. Proceedings. 1999 International Conference on* 12-16 Oct. 1999 Page(s):236 – 244

[14] "Handel-C for Hardware Design", *White Paper*, Celoxica Ltd, August 2002

[15] "BINACHIP-FPGA Datasheet", Binachip Inc., 2005

[16] D. Zaretsky, G. Mittal, X. Tang, P. Banerjee, "Overview of the FREEDOM Compiler for Mapping DSP Software to FPGAs," *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, pp. 37-46, 2004

[17] S. Hauck, T.W. Fry, M.M Hosler, J.P. Kao, "The Chimaera reconfigurable functional unit", *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, Volume 12, Issue 2, Feb. 2004 Page(s):206 – 217

[18] B. Kastrup, "Automatic Synthesis of Reconfigurable Instruction Set Accelerations", PhD Thesis, *Eindhoven University of Technology*, 2003

[19] "Xtensa LX Microprocessor, Overview Handbook", *Tensilica*, Santa Clara, 2004

[20] R. Hartenstein, "Coarse Grain Reconfigurable Architectures", *Proceedings of ASP-DAC, Asia and South Pacific*, 2001

[21] T.J. Callahan, J.R. Hauser, J. Wawrzynek J, "The Garp architecture and C compiler", *IEEE Trans. on Computer*, Volume 33, Issue 4, Page(s):62 – 69, April 2000

[22] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho., "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications", *IEEE Trans. on Computers*, 49(5):465–481, May 2000.

[23] P.M. Heysters, G.K. Rauwerda, T. Lodewijk, G.J.M. Smit, "A Flexible, Low Power, High Performance DSP IP Core for Programmable Systems-on-Chip", *proceedings IP/SOC 2005*, December 7-8, 2005, Grenoble, France

[24] P.M. Heysters, G.K. Rauwerda, G.J.M. Smit, "Implementation of a HiperLAN/2 receiver on the reconfigurable Montium architecture"; *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 26-30 April 2004 Page(s):147

[25] "Avispa-CH1, Communication Signal Processor", *databrief*, SiliconHive, Eindhoven, 2005

[26] "XPP64-A1 Reconfigurable Processor", *Preliminary Datasheet*, PACT XPP Technologies, Munich, 2003.

[27] T. Miyamori, U. Olukotun, "REMARC: Reconfigurable Multimedia array coprocessor", *ACM International Symposium on FPGA*, Monterey, CA, Feb 1998

[28] E. Mirsky and A. DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources," *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'96)*, pp. 157–166, 1996.

[29] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. R. Taylor, "PipeRench: A virtualized programmable datapath in 0.18 micron technology", *In Proc. of IEEE Custom Integrated Circuits Conference*, 2002

[30] 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," *Proc.of Field-Programmable Logic and Applications*, 2003, pp. 61–70.

[31] J. Babb, M. Frank, V. Lee, E.Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agrawal, "The RAW Benchmark Suite: Computation Structures for General-

Purpose Computing, ", *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 97*, 1997, pp. 134–143.

[32] D. Wentzlaff , A. Agarwal , "A Quantitative Comparison of Reconfigurable, Tiled, and Conventional Architectures on Bit-level Computation", *MIT/LCS Technical Report* LCS-TR-944, April 2004

[33] D. C. Cronquist, P. Franklin C. Fisher M. Figueroa and C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths", *Twentieth Anniversary Conference on Advanced Research in VLSI*, 1999

[34] Ebeling, C.; Fisher, C.; Guanbin Xing; Manyuan Shen; Hui Liu, "Implementing an OFDM receiver on the RaPiD reconfigurable architecture", *Computers, IEEE Transactions on* Volume 53, Issue 11, Nov. 2004 Page(s):1436 – 1448

[35] A. Abnous, J. M. Rabaey. "Ultra-low-power domain-specific multimedia processors", *IEEE Transactions on VLSI Signal Processing,* 1996

[36] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, J. M. Rabaey, "A 1-V Heterogeneous Reconfigurable DSP IC for Wireless Baseband Digital Signal Processing", *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, November 2000

[37] H. Zhang, M. Wan, V. George, J. Rabaey, "Interconnect architecture exploration for low-energy configurable single-chip DSPs," *IEEE Computer Society Workshop on VLSI*, pp. 2-8, 1999. 24

[38] P. Hamalainen, J. Heikkinen, M. Hannikainen, T.D. Hamalainen, "Design of Transport Triggered Architecture Processors for Wireless Encryption", *Digital System Design, 2005. Proceedings. 8th Euromicro Conference on* 30 Aug.-3 Sept. 2005 Page(s):144 – 152

[39] J. Heikkinen, J. Sertamo, T. Rautiainen, J. Takala, "Design of transport triggered architecture processor for discrete cosine transform"; *ASIC/SOC Conference, 2002. 15th Annual IEEE International* 25-28 Sept. 2002 Page(s):87 – 91

[40] T. Ishihara, S. Kondou, H. Fukuda, "Low Power Consumption Digital Signal Processor: Hi-Perion", *FUJITSU Science and Technology* vol. 36, pp. 56-62. 2000

[41] D. Lewis, V. Betz, D. Jefferson, A. Lee, C. Lane, P. Leventis, S. Marquardt, C. McClintock, B. Pedersen, G. Powell, S. Reddy, C. Wysocki, R. Cliff and J. Rose, "The Stratix Routing and Logic Architecture," *ACM/Sigda International Symposium on Field-Programmable Gate Arrays*, February 2003, pp. 12 - 20

[42] Betz V., Rose J., and Marquardt A., "Architecture and CAD for Deep-Submicron FPGAs", *Kluwer Academic Publishers*, 1999. ISBN 0-7923-8460-1

[43] Rose J., Brown S., "Flexibility of interconnection structures for field-programmable gate arrays", *Solid-State Circuits, IEEE Journal of* , Vol.26, Iss.3, 1990, Pages: 277-282

[44] J. Rose, R.J. Francis, D. Lewis, and P.Chow, "Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area of Efficiency," *IEEE Journal of Solid-State Circuits,* Vol. 25 No. 5, October 1990, pp. 1217-1225

[45] H. Schmit, V. Chandra, "FPGA switch block layout and evaluation", *ACM International Symposium on FPGA*, Monterey, CA, Feb 2002.

[46] G. Lemieux, D. Lewis, "Circuit Design of Routing Switches", *ACM International Symposium on FPGA*, Monterey, CA, Feb 2002

[47] E. S. Ochotta, P. J. Crotty, C. R. Erickson, C.-T. Huang et al, "A novel predictable segmented FPGA routing architecture", *ACM International Symposium on FPGA*, Monterey, CA, Feb 1998

[48] M. Imran Masud, "FPGA Routing Structures: A novel Switch block and depopulated interconnects matrix architecture", *MASc Thesis*, The University of British Columbia, 1999

[49] M. Imran Masud, S. J.E. Wilton, "A New Switch Block for Segmented FPGAs'", *International Workshop on Field Programmable Logic and Applications*, Aug. 1999

[50] A. El Gamal, J. Greene, J. Reyneri, E. Rogoyski, K. A. El-ayat, A. Mohsen, "An architecture for electrically configurable gate arrays," *IEEE Journal of Solid-State Circuits*, Vol. 24, April 1989, pp.394-398.

[51] Y. W. Chang, D. Wong, and C. Wong, "Universal Switch modules for FPGA design," *ACM Transactions on Design Automation of Electronic Systems*, Vol. 1, January, 1996, pp. 80-101.

[52] G. Lemieux, S.D. Brown, "A detailed router for allocating wire segments in field programmable gate arrays," *Proceedings of the ACM Physical Design Workshop*, April 1993.

[53] S. Wilton, "Architecture and Algorithms for Field-Programmable Gate Arrays with Embedded Memory", PhD thesis, University of Toronto, 1997

[54] Y. Lai, C. Kao, T. Chang, and K. Chen, "A Field Programmable Gate Array Chip with Hierarchical Interconnection Structure," *Proceedings of the 1998 IEEE International Symposium on Circuits and Systems*, Monterey, California, 1998, pp. 402-405.

[55] A.A. Aggarwal, D.M. Lewis, "Routing Architectures for Hierarchical Field Programmable Gate Arrays," *Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Cambridge, Massachusetts, 1994, pp. 475-478.

[56] V.C. Chan , D.M. Lewis, "Area-Speed Tradeoffs for Hierarchical Field-Programmable Gate Arrays," *ACM Fourth International Symposium on Field-Programmable Gate Arrays*, New York, 1996, pp.51-57

[57] V. Betz, J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research", *International Conference on Field Programmable Logic and Applications (FPL) 1997*, pp. 213-222

[58] S.J.E. Wilton, "Embedded memory in FPGAs: recent research results", *Communications, Computers and Signal Processing, 1999 IEEE Pacific Rim Conference on* , 1999 , Page(s): 292 -296

[59] S. Philips, S. Hauck, "Automatic layout of domain-specific reconfigurable subsystems for system-on-a-chip", *ACM International Symposium on FPGA*, Monterey, CA, Feb 2002

ISO/IEC, "MPEG-4 Standard - Visual", ISO/IEC 14496-2, Geneva,

[60] Li Reoxiang Li, Bing Zeng, M.L Liou, "A new three-step search algorithm for block motion estimation", *Circuits and Systems for Video Technology, IEEE Transactions on*, Volume 4, Issue 4, Aug. 1994 Page(s):438 - 442

[61] K.R. Namuduri, Ji Aiyuan, "Computation and performance trade-offs in motion estimation algorithms", *Information Technology: Coding and Computing, 2001. Proceedings. International Conference on*, 2-4 April 2001 Page(s):263 - 267

[62] T. Zahariadis, D. Kalivas, "Fast algorithms for the estimation of block motion vectors", *Electronics, Circuits, and Systems, 1996. ICECS '96., Proceedings of the Third IEEE International Conference on*, Volume 2, 13-16 Oct. 1996 Page(s):716 - 719 vol.2

[63] Shan Zhu, Kai-Kuang Ma, "A new diamond search algorithm for fast block matching motion estimation", *Information, Communications and Signal Processing, 1997. ICICS., Proceedings of 1997 International Conference on*, Volume 1, 9-12 Sept. 1997 Page(s):292 - 296 vol.1

[64] T. Zahariadis, D. Kalivas, "Fast algorithms for the estimation of block motion vectors", *Electronics, Circuits, and Systems, 1996. ICECS '96., Proceedings of the Third IEEE International Conference on*, Volume 2, 13-16 Oct. 1996 Page(s):716 - 719 vol.2

[65] T. Enomoto, A. Kotabe, "A fast motion estimation algorithm and low-power 0.13-μm CMOS motion estimation circuits", *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, Volume 2, 6-9 May 2001 Page(s):449 - 452 vol. 2

[66] Hsien-Hsi Hsieh, Yong-Kang Lai, "A novel fast motion estimation algorithm using fixed subsampling pattern and multiple local winners search", *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, Volume 2, 6-9 May 2001 Page(s):241 - 244 vol. 2

[67] J.W. Suh, Jechang Jeong, "Fast sub-pixel motion estimation techniques having lower computational complexity", *Consumer Electronics, IEEE Transactions on*, Volume 50, Issue 3, Aug. 2004 Page(s):968 - 973

[68] Zhong-Li He; Kai-Keung Chen; Chi-Ying Tsui; N.L. Liou, "Low power motion estimation design using adaptive pixel truncation", *Low Power Electronics and Design, 1997. Proceedings., 1997 International Symposium on*, 18-20 Aug 1997 Page(s):167 - 172

[69] A. Takagi, S. Muramatsu, H. Kiya, "Motion estimation with power scalability and its VHDL model", *Image Processing, 2000. Proceedings. 2000 International Conference on*, Vol.3, 2000, Pages: 118- 121 vol.3

[70] L. Fanucci, R. Saletti, L. Bertini, P. Moio, S. Saponara, "High-Throughput, Low Complexity, Parametrizable VLSI Architecture for Full Search Block Matching Algorithm", *Electronics, Circuits and Systems, 1999. Proceedings of ICECS '99. The 6th IEEE International Conference on* , Vol.3, 1999, Pages: 1479- 1482 vol.3

[71] Yuan-Hau Yeh; Chen-Yi Lee, "Scalable VLSI Architectures For Full-Search Block Matching Algorithms", *Image Processing, 1996. Proceedings., International Conference on* , Vol.1, 1996, Pages: 1035- 1038 vol.2

[72] Xiao-Dong Zhang; Chi-Ying Tsui, "An Efficient And Reconfigurable VLSI Architecture For Different Block Matching Motion Estimation Algorithms", *Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on*, Vol.1, 1997, Pages: 603- 606 vol.1

[73] W. Burleson, P. Jain, S. Venkatraman, "Dynamically parameterized architectures for power-aware video coding: motion estimation and DCT", *Digital and Computational Video, 2001. Proceedings. Second International Workshop on* , Vol., 2001, Pages: 4- 12

[74] H.-J. Stolberg, M. Berekovic, P. Pirsch, H. Runge, H. Moller, J. Kneip, "The M-PIRE MPEG-4 codec DSP and its macroblock engine", *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, Volume 2, 28-31 May 2000 Page(s):192 - 195 vol.2 ,

[75] T. Kumura, D. Ishii, M. Ikekawa, I. Kuroda, M. Yoshida, "A low-power programmable DSP core architecture for 3G mobile terminals", *Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP '01). 2001 IEEE International Conference on*, Volume 2, 7-11 May 2001 Page(s):1017 - 1020 vol.2

[76] M. Berekovic, H.-J. Stolberg, P. Pirsch, H. Runge, "A programmable co-porcessor for MPEG-4 video", *Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP '01). 2001 IEEE International Conference on*, Volume 2, 7-11 May 2001 Page(s):1021 - 1024 vol.2

[77] D. Brash, "The ARM Architecture Version 6 (ARMv6)", White paper, ARM Ltd, January 2002

[78] L. De Vos, M. Stegherr, "Parameterizable VLSI architectures for the full-search block-matching algorithm", *IEEE Transactions on Circuits and Systems*, Vol.36 Issue: 10 , Oct. 1989

[79] L. De Vos, M. Stegherr, T.G. Noll, "VLSI architectures for the full-search blockmatching algorithm," *Acoustics, Speech, and Signal Processing, 1989. ICASSP-89., 1989 International Conference on*, 23-26 May 1989 Page(s):1687 - 1690 vol.3

[80] L. De Vos, M. Schobinger, "VLSI architecture for a flexible block matching processor", *Circuits and Systems for Video Technology, IEEE Transactions on*, Volume 5, Issue 5, Oct. 1995 Page(s):417 - 428

[81] T. Komarek, P. Pirsch, "Array architectures for block matching algorithms", *IEEE Transactions on Circuits and Systems*, Vol. 36 Issue: 10 , Oct. 1989

[82] K.-M. Yang, M.-T. Sun, L. Wu, "A family of VLSI designs for the motion compensation block-matching algorithm", *IEEE Transactions on Circuits and Systems*, Vol. 36 Issue: 10 , Oct. 1989

[83] H. Yeo, Y.H. Hu, "A novel matching criterion and low power architecture for real-time block based motion estimation", *Application Specific Systems, Architectures and*

Processors, 1996. ASAP 96. Proceedings of International Conference on, 19-21 Aug. 1996 Page(s):122 - 130

[84] Hae-Kwan Jung, Chun-Pyo Hong, Jin-Soo Choi, Yeong-Ho Ha, "A VLSI architecture for the alternative subsampling-based block matching algorithm", *Consumer Electronics, IEEE Transactions on*, Volume 41, Issue 2, May 1995 Page(s):239 - 247

[85] D. Xu, J.M. Noras, W. Booth, "A simple and efficient VLSI architecture for a very fast high performance three step search algorithm", *High Performance Architectures for Real-Time Image Processing, IEE Colloquium on*, 12 Feb. 1998 Page(s):6/1 - 6/6

[86] Hangu Yeo, Yu Hen Hu, "A novel modular systolic array architecture for full-search block matching motion estimation", *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, Volume 5, 9-12 May 1995 Page(s):3303 - 3306 vol.5

[87] Bo-Sung Kim, Jun-Dong Cho, "VLSI architecture for low power motion estimation using high data access reuse", *ASICs, 1999. AP-ASIC '99. The First IEEE Asia Pacific Conference on,* 23-25 Aug. 1999 Page(s):162 - 165

[88] Sung Bum Pan, Seung Soo, Chae Rae-Hong Park, "A novel VLSI architecture for the full search block matching algorithm using systolic array", *Circuits and Systems, 1996. ISCAS '96., 'Connecting the World'., 1996 IEEE International Symposium on*, Volume 2, 12-15 May 1996 Page(s):750 - 753 vol.2

[89] S. Kittitornkun, Hu Yu Hen, "Frame-level pipelined motion estimation array processor", *Circuits and Systems for Video Technology, IEEE Transactions on*, Volume 11, Issue 2, Feb 2001 Page(s):248 - 251

[90] N. Ahmed, T. Natarjan, K.R. Rao, "Discrete Cosine Transform", *IEEE Transactions on Computers*, vol. 23, 1974, pp .90-93

[91] W. Chen, C. H. Smith, S. Fralick, "A fast computation algorithm for the discrete cosine transform", *IEEE Transactions on Communications*, vol. 25, pp. 1004–1009, September 1977

[92] C. Loffer, A. Ligtenberg, G. S. Moschytz, "Practical fast 1-D DCT algorithm with 11 multiplications", *Proceedings of ICASSP*, vol.2 pp. 988-991, 1989

[93] Sungwook Yu; Swartziander, E.E., Jr., "DCT implementation with distributed arithmetic", *IEEE Transactions on Computers* , Vol. 50 Issue 9 , Sept. 2001

[94] Chin-Liang Wang; Chang-Yu Chen, "High-throughput VLSI architectures for the 1-D and 2-D discrete cosine transforms", *IEEE Transactions on Circuits and Systems for Video Technology*, Volume: 5 Issue: 1 , Feb. 1995

[95] Yu-Tai Chang; Chin-Liang Wang; Ching-Hsien Chang, "A new systolic architecture for fast DCT computation', *IEEE International Symposium on Circuits and Systems*, 1996. ISCAS '96., vol. 2 , 1996

[96] J.E Volder, "The CORDIC trigonometric computing technique", *IRE Trans. On Electronic Computers*, Sept. 1959

[97] Feng Zhou, P. Kornerup, "High speed DCT/IDCT using a pipelined CORDIC algorithm", *Proceedings of the 12th Symposium on Computer Arithmetic*, 1995

[98] E. P. Mariatos, D. E. Metafas, J.A. Hallas, C.E. Goutis, "A fast DCT processor, based on special purpose CORDIC rotators", *Proc. IEEE Int. Symposium. Circuits Systems*, vol. 4, 1994

[99] Yang, K.-M.; Sun, M.-T.; Wu, L. , "A family of VLSI designs for the motion compensation block-matching algorithm", *IEEE Transactions on Circuits and Systems*, Vol. 36 Issue: 10 , Oct. 1989

[100] M.A. BenAyed, L. Dulau, P. Nouel, Y. Berthournieu, N. Masmoudi, P. Kadionik, L. Kamoun, "New design using a VHDL description for DCT based circuits", *Proceedings of the Tenth International Conference on Microelectronics*, ICM '98. , 1998

[101] Kyeounsoo Kim; Jong-Seog Koh, "An area efficient DCT architecture for MPEG-2 video encoder", *Consumer Electronics, IEEE Transactions on* , vol. 45 Issue: 1 , Feb. 1999

[102] B.L. Jian, Z. Xuan, T.J. Rong, L. Yue, "An efficient VLSI architecture for 2D-DCT using direct method", *Proceedings. 4th International Conference on ASIC*, 2001

[103] J. Prado, P. Duhamel, "A polynomial transform based computation of the 2D DCT with minimum multiplicative complexity", *ICASSP* 1996

[104] Nam Ik Cho; San Uk Lee, "Fast algorithm and implementation of 2-D discrete cosine transform", *IEEE Transactions on Circuits and Systems*, Volume: 38 Issue: 3 , March 1991

[105] E. Feig, S. Winograd, "Fast algorithms for the discrete cosine transform", *IEEE Transactions on Signal Processing*, Volume: 40 Issue: 9 , Sept. 1992

[106] Shen-Fu Hsiao; Wei-Ren Shiue, "A new hardware-efficient algorithm and architecture for computation of 2-D DCTs on a linear array", *IEEE Transactions on Circuits and Systems for Video Technology*, Volume: 11 Issue: 11 , Nov. 2001

[107] Yi Yang; Chunyan Wang; Omair Ahmad, M.; Swamy, M.N.S., "An on-line CORDIC based 2-D IDCT implementation using distributed arithmetic", *Sixth International Symposium on Signal Processing and its Applications*,. 2001 , Vol. 1

[108] Y. Arai, T. Agui, and M. Nakajima, "A fast DCT-SQ scheme for images", *The Transactions of the IEICE*, vol. E71, pp. 1095-1097, November 1988.

[109] S. Baloch, "High Performance, Reconfigurable Low Power SoC Architectures For Mobile Platforms", MSc Thesis, ISLI/University of Edinburgh, Livingston, 2003

[110] L. Zhenyu, S. Khawam, T. Arslan, A. Erdogan,; "A Low Power Heterogeneous Reconfigurable Architecture For Embedded Generic Finite State Machines"; *Proceedings of SOC Conference, 2005. IEEE International* 25-28 Sept. 2005

[111] D. Wentzlaff, "Architectural Implications of Bit-level Computation in Communication Applications", *MSc Thesis*, Massachusetts Institute of Technology 2002

[112] G. Lemieux, D. M. Lewis, "Circuit design of routing switches", *ACM International Symposium on FPGA*, Monterey, CA, Feb 2002: 19-28

[113] S. Agarwala, et al, "A 600-MHz VLIW DSP", *IEEE Journal of Solid-State Circuits*, Vol. 37, Iss. 11, Nov. 2002, pp. 1532-1544

[114] G. Martinez, "TMS320VC5501/02 Power Consumption Summary", *Application Report, TI*, SPRAA48, July 2004

[115] "ARM7 Thumb Family Datasheet", *ARM DOI 0035-3/02.02*, ARM Ltd, 2002

[116] "ARM9 Family Datasheet", ARM DOI 0034-4/06.02, ARM Ltd, 2002

[117] OpenRISC, http://www.opencores.org/projects.cgi/web/or1k

[118] GNU C compiler, 4.0, http://gcc.gnu.org/ 2005

[119] "TMS320C5000 CPU and Instruction Set Reference Guide", Texas Instruments, October 2000

[120] G. Martinez, "TMS320VC64010/13 Power Consumption Summary", *Application Report, TI*, SPRAA50, September 2002

[121] "TIC6000 Compiler Benchmarks", Texas Instruments, 2004

[122] "Framing Structure, Channel Coding and Modulation for Digital Terrestrial Television", *DVB Document A012*, DVB Project Office, Geneva, Switzerland, June 1996

[123] "ZL10353 Datasheet Fully Compliant NorDig Unified COFDM Digital Terrestrial TV (DTV) Demodulator", *Datasheet*, Zarlink, Ottawa, 2005

[124] Wang, Chua-Chin; Huang, Jian-Ming; Cheng, Hsian-Chang , "A 2K/8K Mode Small-Area FFT Processor for OFDM Demodulation of DVB-T Receivers", *Consumer Electronics, IEEE Transactions on*, Volume 51, Issue 1, Feb. 2005 Page(s):28 - 32

[125] J.W. Cooley, J.W. Tukey, "An algorithm for the machine calculation of complex Fourier series", *Math. Comput.* 19:297-301, 1965

[126] "MAD: MPEG Audio Decoder", libmad, Underbit Technologies, San Diego, 2005, http://www.underbit.com/products/mad/

[127] ffmpeg library, http://ffmpeg.sourceforge.net/

[128] I. Nousias, "Path-Encoding. An efficient representation of netlists and code compression technique for Direct Network-based RCs", *Internal Document*, University of Edinburgh, August, 2005

[129] I. Nousias, "Reducing data-memory access by using sub-step time tags", *Internal Document*, University of Edinburgh, April, 2005

[130] ISO/IEC, "MPEG-4 Standard - Visual", *Specification*, ISO/IEC 14496-2, Geneva

[131] G. E. Moore, "Cramming more components onto integrated circuits", Electornics, vol. 38, pp. 114-117, 1965

[132] K. Compton, S. Hauck. Totem, "Custom Reconfigurable Array Generation", *9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01).*

[133] Murray, A.F.; Denyer, P.B., "A CMOS Design Strategy for Bit-Serial Signal Processing", *Solid-State Circuits, IEEE Journal of* Volume 20, Issue 3, Jun 1985 Page(s):746 – 753