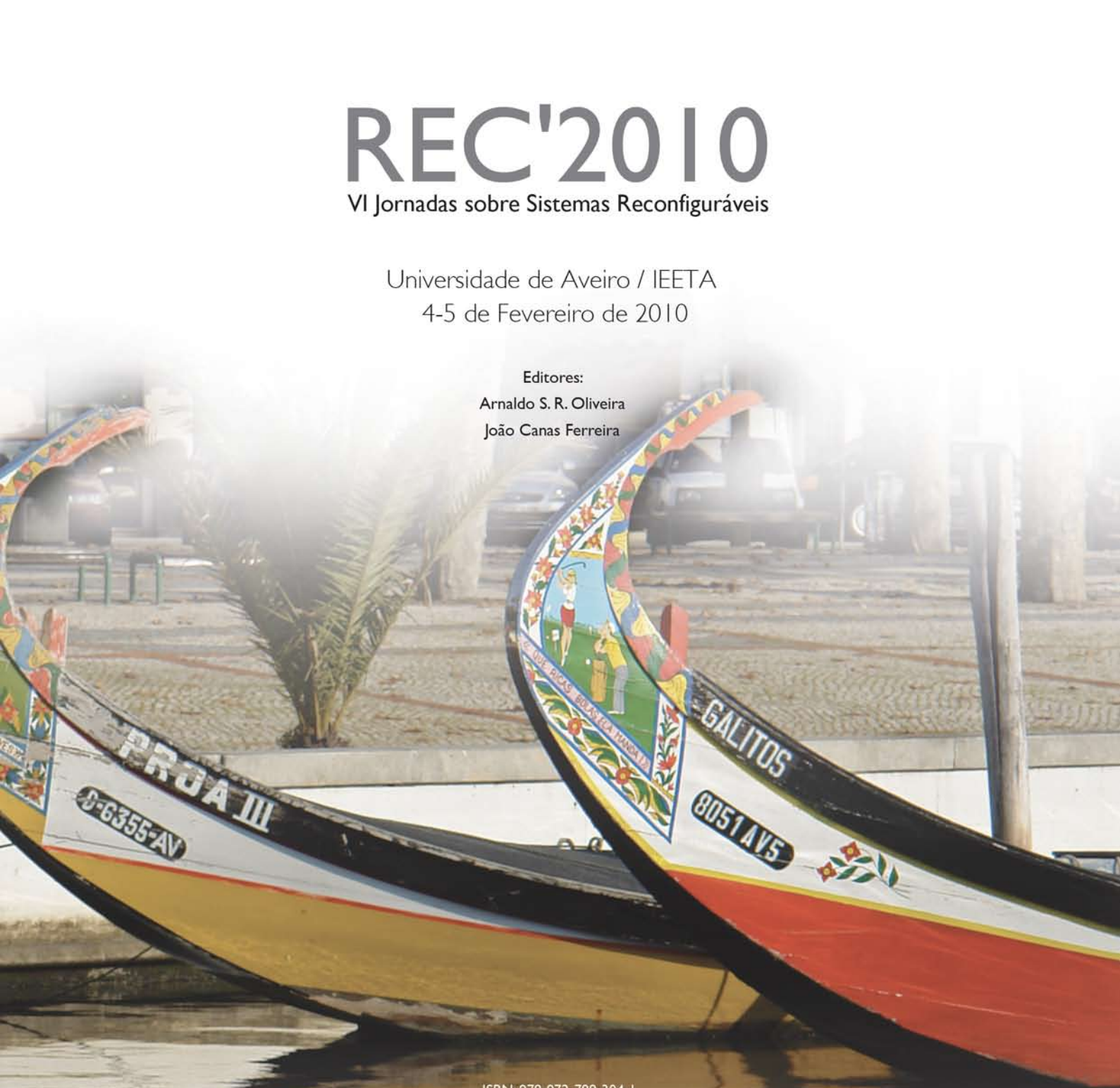
REC'2010

VI Jornadas sobre Sistemas Reconfiguráveis

Universidade de Aveiro / IEETA
4-5 de Fevereiro de 2010

Editores:
Arnaldo S. R. Oliveira
João Canas Ferreira



ISBN: 978-972-789-304-1

organização

universidade de aveiro

instituto de engenharia electrónica e telemática de aveiro
ieeta

Universidade do Porto
FEUP Faculdade de Engenharia

apoios e patrocínios

FCT Fundação para a Ciência e a Tecnologia

Região Centro

IEEE PORTUGAL SECTION

CAS IEEE CIRCUITS AND SYSTEMS SOCIETY

CONSUMER ELECTRONICS SOCIETY

PT INOVAÇÃO

XILINX

REC'2010

Actas das

VI Jornadas sobre Sistemas Reconfiguráveis

4 e 5 de Fevereiro de 2010

Universidade de Aveiro

IEETA

Editores:
Arnaldo S. R. Oliveira
João Canas Ferreira

Edição: Universidade de Aveiro – Comissão Organizadora da REC'2010
Arnaldo S. R. Oliveira e João Canas Ferreira

N. DL: 304580/10
ISBN: 978-972-789-304-1

Design da Capa: Sérgio Cabaço
Impressão e Acabamentos: Designeed, Lda

Conteúdo

Sessão Posters

Prefácio

As VI Jornadas sobre Sistemas Reconfiguráveis decorrem na Universidade de Aveiro, nos dias 4 e 5 de Fevereiro de 2010. Esta edição dá continuidade à série de eventos iniciada na Universidade do Algarve, em 2005, com edições anuais posteriores na Faculdade de Engenharia da Universidade do Porto, no Instituto Superior Técnico da Universidade Técnica de Lisboa, no Departamento de Informática da Universidade do Minho e na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa. As Jornadas têm conseguido constituir-se como o ponto de encontro anual para a comunidade científica de língua portuguesa com reconhecida actividade de investigação e desenvolvimento na área dos sistemas electrónicos reconfiguráveis.

O programa das VI Jornadas – REC'2010 – tem uma organização semelhante às edições anteriores, decorrendo durante dia e meio. Este ano, as Jornadas incluem duas apresentações convidadas proferidas por Leonel Sousa, professor do IST e colega com ligações à série de Jornadas REC desde o seu início, e por Chris Berg, engenheiro da MIPS Technologies e responsável na Europa pelas relações com os clientes e suporte técnico desta empresa. A ambos agradecemos a disponibilidade para partilharem com os participantes da REC'2010 as suas experiências e conhecimentos.

O programa conta ainda com a apresentação de 25 comunicações regulares nas áreas de: linguagens e algoritmos, telecomunicações, aplicações científicas, arquitectura e circuitos aritméticos, instrumentação e controlo, e processamento de áudio/vídeo. Estas contribuições correspondem a 11 artigos longos, 11 artigos curtos e 3 posters, todos aprovados para publicação e apresentação pelo Comité Científico. De referir que das 78 revisões recebidas (correspondentes a uma média de cerca de 3 revisões por artigo), 62% foram produzidas com um grau de confiança de "alto" ou "especialista", permitindo confirmar o elevado grau de adequabilidade dos membros do Comité Científico para avaliar os trabalhos submetidos.

A organização destas Jornadas contou com o apoio de diversas pessoas, entidades e empresas, em relação às quais gostaríamos de expressar o nosso agradecimento. Em primeiro lugar devemos um agradecimento especial aos autores que contribuíram com os trabalhos incluídos nestas Actas, bem como aos membros do Comité Científico pelo excelente trabalho produzido, concretizado em revisões correctas e que, estamos certos, permitiram melhorar a qualidade dos trabalhos submetidos.

Os nossos agradecimentos ao Instituto de Engenharia Electrónica e Telemática de Aveiro e à Universidade de Aveiro, pela imprescindível colaboração na organização, apoio técnico e administrativo, e por terem cedido um espaço muito adequado para a realização das Jornadas.

Em termos de apoio financeiro e patrocínios destacamos os concedidos pela Fundação para a Ciência e a Tecnologia (Fundo de Apoio à Comunidade Científica), pelo IEEE (Portugal Section e CE/CAS/BT Chapter), pela PTInovação e pela Xilinx. Finalmente, uma palavra de agradecimento pelo valioso apoio dado à organização por parte de vários colegas, funcionários, bolseiros e estudantes da UA. Bem hajam.

Esperamos que esta edição das Jornadas constitua, uma vez mais, um espaço para divulgação e discussão dos trabalhos apresentados, bem como de convívio aberto a todos quantos partilham interesses na área dos sistemas electrónicos reconfiguráveis.

Arnaldo S. R. Oliveira, Universidade de Aveiro – DETI / IEETA
João Canas Ferreira, Faculdade de Engenharia da Universidade do Porto / INESC Porto
Fevereiro 2010

Comissão Organizadora

Arnaldo S. R. Oliveira
Universidade de Aveiro – DETI
Instituto de Engenharia Electrónica e Telemática de Aveiro

João Canas Ferreira
Faculdade de Engenharia da Universidade do Porto
INESC Porto

Secretariado e Apoio Local

Anabela Viegas
Instituto de Engenharia Electrónica e Telemática de Aveiro

Nelson Silva
Universidade de Aveiro

Contacto Geral

Organização da REC'2010
Instituto de Engenharia Electrónica e Telemática de Aveiro
Universidade de Aveiro
Campus Universitário de Santiago
3810-193 Aveiro
Portugal
Tel.: +351 234 370 500
Fax: +351 234 370 545
E-mail: rec2010@ieeta.pt
URL: http://www.ieeta.pt/rec2010/

Apoios e Patrocínios

Comité Científico

Coordenação

Arnaldo S. R. Oliveira	Universidade de Aveiro / IEETA
João Canas Ferreira	Fac. de Engenharia da Univ. do Porto / INESC Porto

Comité de Programa

Ana Antunes	Instituto Politécnico de Setúbal
António Esteves	Universidade do Minho
António Ferrari	Universidade de Aveiro / IEETA
António Valente	Universidade de Trás-os-Montes e Alto Douro
Fernando Gonçalves	Instituto Superior Técnico / INESC-ID
Helena Ramos	Instituto Superior Técnico / IT
Helena Sarmento	Instituto Superior Técnico / INESC-ID
Hélio Mendonça	Fac. de Engenharia da Univ. do Porto / INESC Porto
Henrique Santos	Universidade do Minho
Horácio Neto	Instituto Superior Técnico / INESC-ID
Iouliia Skliarova	Universidade de Aveiro / IEETA
João M. P. Cardoso	Fac. de Engenharia da Univ. do Porto / INESC Porto
João Lima	Universidade do Algarve
João M. Fernandes	Universidade do Minho
José Augusto	Fac. de Ciências da Univ. de Lisboa / INESC-ID
José Carlos Alves	Fac. de Engenharia da Univ. do Porto / INESC Porto
José C. Metrôlho	Instuto Politécnico de Castelo Branco
Leonel Sousa	Instituto Superior Técnico / INESC-ID
Luís Cruz	Universidade de Coimbra
Luís Gomes	Universidade Nova de Lisboa / UNINOVA
Luís Nero	Universidade de Aveiro / IT
Manuel Gericota	Instituto Superior de Engenharia do Porto
Mário Calha	Fac. de Ciências da Univ. de Lisboa / LaSIGE
Mário Véstias	Instituto Superior de Engenharia de Lisboa / INESC-ID
Morgado Dias	Universidade da Madeira
Nuno Roma	Instituto Superior Técnico / INESC-ID
Paulo Flores	Instituto Superior Técnico / INESC-ID
Paulo Teixeira	Instituto Superior Técnico / INESC-ID
Pedro Diniz	Instituto Superior Técnico / INESC-ID
Ricardo Machado	Universidade do Minho
Rui Aguiar	Universidade de Aveiro / IT
Valeri Skliarov	Universidade de Aveiro / IEETA

Comunicações Convidadas

Moderação: João Canas Ferreira
Fac. de Engenharia da Univ. do Porto / INESC Porto

Foreseeing the Role of Reconfiguration in Multi-core Architectures

Leonel Sousa
Instituto Superior Técnico/INESC-ID
las@inesc-id.pt

Extended Abstract

As it is known, in the last twenty years processors have evolved both by exploiting instruction level parallelism at the micro-architecture level and by increasing its operating frequency. However, power thermal issues have become a serious constraint in the development of improved micro-architectures, and thus have been limiting the performance rate growth of such micro-architectures. Multi-core processors have been introduced as a solution to sustain the performance increase rate and, at the same time, to keeping the design within the required power budget. While currently general-purpose processors are available with up to tens of cores, probably hundreds of cores will be available in the near future [1]. Moreover, currently available multi-core chips are homogeneous, but, in order to improve the efficiency for different applications, in the future processors will have to offer cores with different characteristics and architectures [2]. Some features of these architectures may be dynamically configured at a coarser-grain level, namely by morphing the processor within a set of possible configurations. Reconfiguration includes basic techniques, namely for adjusting the operating frequency and the number of active cores, as well as more complex dynamic mechanisms to configure the memory and the micro-architecture [3][4].

This talk addresses the opportunities and challenges of reconfiguration to adapt particular features of multi-core architectures. Apart from discussing the work that has been done so far, we also foresee the future of efficient multi-core processors: they will have the ability to dynamically morph to different configurations, and virtualization will play an important role. A software virtualization layer can be used as a wrapper to hide the complexity of this hardware reconfiguration, as well as to monitor the requirements of the applications. Besides orchestrating the execution of the applications given the available multi-cores and their characteristics, this software layer should also adapt the hardware to their demands. Fig.1 shows that the foreseeing reconfigurable architecture requires a dedicated reconfiguration controller, which is implemented as the Reconfiguration core used to support this Service [5]. Moreover, the available reconfiguration capabilities can be described by considering separately the computational elements and the memory hierarchy.

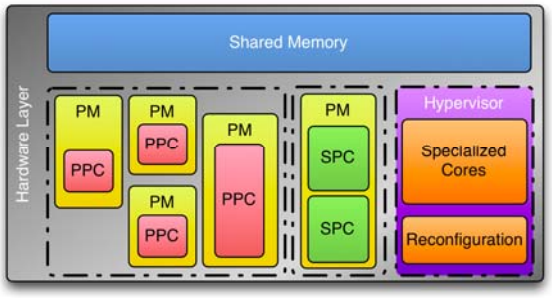


Fig. 1: Morphable Architecture and its Components: Private Memory (PM), Sequential Processing Core (SPC), Parallel Processing Core (PPC).

Acknowledgements

Part of the work presented in this talk has resulted from collaborative research work performed at INESC-ID/IST and at the University of Cyprus, with Prof. Pedro Trancoso, Panayiotis Petrides and Frederico Pratas.

References

[1] "From a Few Cores to Many: A Tera-scale Computing Research Overview", White Paper, Intel, 2006; ftp://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf.
[2] M. Hill and M. Marty, "Amdahl's Law in the Multicore Era", IEEE Computer, 2008, pp. 33-38.
[3] K. Sankaralingam, R. Nagarajan *et al*., "TRIPS: A Polymorphous Architecture for Exploiting ILP, TLP, and DLP", ACM Transactions on Architecture and Code Optimization, vol. 1, issue 1, 2004, pp. 62-93.
[4] E. Ipek, M. Kirman, M. Nevin and J. Martinez, "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors", 34th International Symposium Computer Architecture, 2007, pp. 186-197.
[5] F. Pratas, P. Petrides, P. Trancoso and L. Sousa, "Virtualization for Teams-of-Morphable Cores", INESC-ID Technical Report, 2009.

MIPS® IP cores and the SEAD-3 FPGA-based Development Platform and Tools

Chris Berg
MIPS Technologies
chrisb@mips.com

Extended Abstract

MIPS Technologies [1] is a leading provider of industry-standard processor architectures and cores that power some of the world's most popular products for the home entertainment, communications, networking and portable multimedia markets.

For more than two decades, MIPS Technologies has been a leader and innovator in the worldwide embedded semiconductor market. At the heart of MIPS is its architecture, developed 20 years ago by Stanford University engineering Professor John Hennessy, who is now president of Stanford University. Hennessy took the lead in RISC processing and created an elegant, streamlined architecture with scalability that has met the demands of generations of applications, preserving the wealth of development tools and software that support them. Today, the MIPS® architecture is an industry standard that offers both 32- and 64-bit variants. The MIPS64® architecture is a natural evolution from the MIPS32® base and is upward compatible to 32-bit software on a binary basis. This is very different from other architectures that often have to go long, and odd, ways to add 64-bit capability to their products.

All of today's MIPS cores are synthesizable and highly configurable. As such, they can be adapted to the requirements that exist in the various SoC designs that they are implemented in. The synthesizable nature of the cores enables designers to target the core for high clock speeds, low power consumption or an implementation that balances these two extremes, giving each core a wide usage range. Of course an FPGA implementation is possible, giving designers the opportunity to build an evaluation system before actual silicon is available. This can speed up verification as well as software and firmware development.

As an FPGA implementation of the cores can be updated easily, it is also a perfect vehicle for the development of CorExtend instructions. MIPS' CorExtend technology gives designers the ability to add custom instructions to the core.

The core pipeline automatically recognizes these instructions, reads the register file and passes the values to the CorExtend block. After processing the results are transferred back to the pipeline for insertion into the register file. Dependencies are automatically checked making it easy to implement new instructions in very few logic gates. In benchmarks, CorExtend shows the potential for dramatic speed increases, sometimes in order of magnitudes.

The SEAD-3 development board is designed to support all of these efforts. The core can be contained in the on-board FPGA or in a separate module if the main FPGA is used for additional system logic. There are plenty of peripherals already included on the board such as UARTs, Ethernet, LCD Display, Flash, GPIO, DRAM and others. The SEAD-3 board also provides for user connectors, making it easy to add more functionality through daughter cards.

There are many different sources for development and debugging tools, both commercial and open source. MIPS supports the GNU compiler toolchain as well as Linux [2]. Many other RTOSs have been ported to MIPS, so that finding the right fit should not be difficult.

References

[1] http://www.mips.com
[2] http://www.linux-mips.org

Sessão Regular 1

Linguagens e Algoritmos

Moderação: Helena Sarmento
Instituto Superior Técnico / INESC-ID

Algorithms for run-time placement and routing on Virtex II Pro FPGAs*

Miguel L. Silva
DEEC, Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias,
4200-465 PORTO, Portugal
mlms@fe.up.pt

João Canas Ferreira
INESC Porto, Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias,
4200-465 PORTO, Portugal
jcf@fe.up.pt

Abstract

Run-time reconfiguration is a useful approach to the implementation of highly-adaptive embedded systems. To generate partial bitstreams at run-time for dynamic reconfiguration of sections of a platform FPGA we combine partial bitstreams of coarse-grained components specified by an acyclic netlist. The placement an routing algorithm play and essential role on the generation of partial bitstreams. A greedy placement heuristic based on topological sorting is used to determine the positions of individual components, and a router based on non-backtracking search over restricted areas determines the routes for the interconnections. The approach is validated with a set of 35 benchmarks (both synthetic and application-derived) having between three and 41 components, the complete process of bitstream generation takes between 7 s and 101 s (average 48.3 s) when running on an embedded PowerPC 405 microprocessor clocked at 300 MHz.

1. Introduction

This paper proposes a method to generate partial bitstreams at run-time in order to partially reconfigure an FPGA. The hardware infrastructure is assumed to include a microprocessor for running the bitstream creation procedure, and to load the newly created bitstream to a specific FPGA area without disturbing the operation of other parts of the system.

For the specific implementation described here, the program runs on an embedded processor in the same FPGA that is being reconfigured. Highly adaptive embedded systems may employ the creation of partial bitstreams at run-time in situations where it is impractical to create all necessary bitstreams at design time, either because there are too many possibilities (e.g., shape-adaptive video processing [1]), or because the required information is only available at run-time (e.g., self-adaptive systems [2]).

The proposed approach is based on placing medium-sized components (like adders, comparators, and multipliers) in a reserved area, and then routing the interconnections among the components, and between the components and the area's I/O terminals. Since platform FPGAs have a heterogeneous fabric (with, e.g., RAM blocks and dedicated multipliers), information about the relative position of resources in the component is required to determine whether a specific location is compatible. For routing purposes, components are treated as black boxes with I/O pins at the periphery. The final partial bitstream is created by merging the component bitstreams (after relocation) into the bitstream for the empty reserved area, and then by further modifying the result to include the connections determined in the routing phase.

Because placement and routing must be performed in a resource-limited context, simple algorithms are employed with the purpose of obtaining acceptable solutions in a reasonable time. Placement is done by a greedy strategy based on sorting the components in topological order. Routing is performed by finding the shortest path from a source terminal to the target terminal for successive nets; target terminals belonging to the same net can share routing resources. These procedures have been implemented in the C programming language and included in a code library for use by applications that wish to take advantage of this approach to improve system adaptability without foregoing hardware support for compute-intensive routines.

An implementation of the proposed approach was evaluated for synthetic and application-derived benchmarks containing between three and 41 components (average: 15 components). For this set of benchmarks, the whole process of bitstream generation takes between 7 s and 101 s (average 48.3 s) on a PowerPC 405 microprocessor clocked at 300 MHz. Both the hardware organization and the process for component creation process employed in the prototype that was used to collect these results have been previously described in [3]. The hardware platform has a Xilinx Virtex-II Pro device with two embedded PowerPC cores [4], although only one is used for this work. The system has a reserved dynamic area that can be configured through partial bitstreams that are loaded using the internal configuration access port (ICAP).

The rest of the paper is organized as follows. Section 2 describes the context for the research reported in the paper and describes previous work. Section 3 presents the details of the placement and routing tasks, including the simplified resource model adopted in order to reduce execution time.

*First author funded by FCT scholarship SFRH/BD/17029/2004. Work supported by FCT project PTDC/EEA-ELC/69394/2006.

Results for an implementation running on the Virtex-II Pro platform FPGA are reported in section 4 for a set of 35 benchmarks. Finally, section 5 presents the conclusions.

2. Related work

In the context of FPGA-based systems, run-time reconfiguration (RTR) designates the capability to alter the hardware design realized by the FPGA in the course of the execution of an application. Basic RTR requires just fast reconfigurability, typically provided by an SRAM-based FPGA. More effective use of RTR can be made if the FPGA supports partial active reconfiguration, i.e., when sections of the reconfigurable fabric may be re-programmed without affecting other sections. This feature enables the implementation of compact, self-adapting systems.

One of the issues raised by RTR concerns the generation of the required partial configurations. This is commonly done at design time, when all eventually useful partial configurations must be specified and created [5, 6]. Several approaches to the relocation of partial bitstreams have been proposed, including both software tools [7,8] and hardware solutions [9,10]. Bitstream relocation is explicitly included in recent design flows [11].

In all cases, the synthesis tools must be run for each partial configuration, making the generation of partial configurations time-consuming. A solution to this problem based on building new partial bitstreams by combining bitstreams of smaller components is described in [3]. The creation of the new bitstreams requires assigning positions of the reconfigurable area to components (placement), relocating and merging the individual component bitstreams, and interconnecting the components (routing) by modification of the merged bitstream. Because this approach does not rely on the synthesis of logic descriptions, it is a good candidate for implementation in an embedded system for the purpose of creating new dynamically reconfigurable modules (partial bitstreams) at run-time.

Efforts to speed-up placement and routing for FPGAs were initially motivated by applications to logic emulation and custom computing. Trade-offs between area and execution time for placement are discussed in [12], where the authors describe a placer that obtains a 52-fold reduction in execution time for a 33% increase in circuit area. Trade-offs between execution time and critical path delay for placement and routing of FPGA circuits are discussed in [13] . By combining different algorithms for placement and routing, the authors of that work obtained a wide range of solutions, including a 3-fold speedup with a 27% degradation of critical path delay. A router for just-in-time mapping of a device-independent configuration description to a specific device architecture is described in [14] : that router is able to produce good hardware circuits using 13 times less memory and executing 10 times faster than VPR [15].

A channel router for the Wires-on-Demand RTR framework is implemented in [16] . The router uses a simplified resource database that is several orders of magnitude smaller than the one used by vendor tools. It uses simple algorithms to find local routes between blocks using relatively few computational resources. Results obtained with a 2.8 MHz Pentium 4 computer indicate that, compared to vendor tools, memory consumption during execution is three orders of magnitude smaller and execution is four orders of magnitude faster, for an average increase in delay of 15% (over a set of seven small benchmarks).

A simplified version of the bitstream assembly approach of [3] is implemented in [17] for an embedded system with a Virtex-II Pro device. The system used described and evaluated in the next sections is an evolution and extension of that work.

3. Placement and routing

Placement and especially routing are generally very demanding tasks. In order to perform them at run-time in embedded systems, we work with coarse-grained components, and use a simplified model of the resources together with simple, greedy place and route algorithms. The main goal is to find a useful solution rapidly, not to exploit all the available resources optimally.

3.1. Resource models

Placement and routing for island-style FPGAs like the Virtex-II Pro is a resource and time consuming task, in part due to the need to handle a large amount of fine-grained resources. For an embedded system with limited computational resources a more coarse-grained approach is required.

In the approach presented here the basic functional element is a component that takes up a certain area of the FPGA fabric (specified in CLBs). This rectangular-shaped component must have all its terminals on the left or right sides. Physically, the terminals are inputs or outputs of LUTs defined at design time. Typically, components have a functional core between a left column of input CLBs and a right column of output CLBs, although this arrangement is not strictly necessary. Terminals must be located on the borders, because the components are considered as black boxes during placement and routing: no overlap of components is allowed and no routing over components is done.

The simplified placement procedure groups components into vertical stripes. The position of a component inside a stripe and the width of the stripe depend on the physical resources used by the component. Routing is restricted to connections between components in adjacent stripes. This restriction guarantees that routing does not interfere with the rest of the system, reduces the search space, and simplifies the process significantly.

All connections are unidirectional: terminals are either inputs or outputs. The output terminals of one component connect to one or more terminals of other components on the next stripe. The terminals to be connected are typically located in adjacent CLB columns. If there are more columns between them, these columns must be empty. In order to limit the effort during routing, only one additional empty column is currently allowed, to account for constraints imposed by the embedded block RAMs (BRAMs).

Due to the physical arrangement of the reconfigurable fabric, two adjacent stripes may be separated by an unused BRAM column in some cases. The unused BRAM column is considered simply as another set of routing resources.

The Virtex-II Pro FPGA has a segmented interconnection architecture, where segments are connected by a regular array of switch matrices, which are connected between themselves and to the other resources (like CLBs and BRAMs) [18]. A large number of routing resources, grouped in vertical and horizontal channels, connect the switch matrices. In order to simplify routing, only a subset of the available segments is used:

- direct connections (vertical, horizontal and diagonal connections to neighboring CLBs);
- double lines (connections to every first and second CLB in all four directions);
- vertical hex lines (connections to every third or sixth CLB above or below).

Long lines (i.e., bidirectional wires that distribute signals across the full device height and width) are excluded, because they can interfere with circuitry outside of the dynamic area. Horizontal hex lines were excluded because they connect to every third or sixth CLB to the left or the right, and therefore reach beyond the area reserved for routing. It is unnecessary to consider other dedicated routing resources (like carry chains, for instance), because they have no bearing on the connections that are to be established at run-time.

The resulting simplified model of the switch matrix associated with each CLB contains 116 pins, distributed as follows:

- 16 direct connections to the 8 neighboring CLBs;
- 40 double lines: 10 in each of the four directions up, down, left and right;
- 20 vertical hex lines: 10 upwards and 10 downwards;
- 8 connections to the outputs of the 4 slices in the associated CLB;
- 32 connections to the inputs of the 4 slices in the associated CLB.

A switch matrix pin is identified by its index in this list of pins. It is also necessary to keep information on the possible connections from a given pin to other pins of the switch matrix. The information required in this case includes the following data for each target pin:

- identification (index) of the target pin;
- relative vertical distance of the endpoints of the connection starting at the target pin (e.g., +1 and +2 in the case of a double line connection in the up direction);
- relative horizontal distance of the endpoints of the connection starting at the target pin (e.g., -1 and -2 in the case of a double line connection in the left direction).

The algorithm of section 3.3 models the area reserved for routing as a two-dimensional array of switch matrices, and employs a data structure based on the simplified model just described to keep track of resource usage.

Algorithm 1: Greedy level-oriented component placement

```
Data: Netlist N of all components
Result: B: merged bitstream with all components
        R: routing information

L ← LevelAssignment(N)
AddStripeInformation(L)
x ← 0, ℓ ← 0
initialize B to the default bitstream
initialize R
foreach S ∈ L do
    y ← 0
    foreach c ∈ S do                          // ordered scan of S
        y1 ← y + YOffset(c)
        if y1 + Height(c) ≤ DeviceHeight then
            x1 ← x + XOffset(c)
            merge bitstream of component c into B at (x1, y1)
            if
            Width(c) + XOffset(c) < Width(S) ∨ x1 ≠ x ∨ y1 ≠ y
            then
                merge feed-through components in B
            update R with final terminal positions for inserted
            component
            y ← y1 + Height(c)
        else
            fail
        if max(Level(Successors(c))) > ℓ + 1 then
            insert feed-through component in level ℓ + 1 of N
    ℓ ← ℓ + 1
    x ← x + Width(S)
```

3.2. Placement

The main input to the placement phase is a component netlist specifying the components to be used and the unidirectional connections between their terminals. No cycles between the components are allowed, i.e., the netlist must define a directed acyclic graph.

The general approach to placement is to find an arrangement of components in columns, so that directly connected components are adjacent to each other. The arrangement in columns was chosen because it matches the reconfiguration mechanism of Virtex-II-Pro FPGAs, where the smallest unit of reconfiguration data (called a frame) applies to an entire column of resources.

A high level description of the implemented greedy placement approach is shown in Algorithm 1, and two examples of component placement inside a stripe are displayed in figure 1. Positions are specified in terms of CLB rows and columns, with the origin at the top left corner of the device.

The first step of the placement algorithm is to group the components by levels (function `LevelAssignment()`. The first level contains the components whose inputs are connected to the interface of the dynamic area. Second level components have all their input terminals connected to first-level components and so forth. If a component has more than one source, the component will be assigned to the level following the highest-numbered source. This is equivalent to processing the components in topological order.

The next step is to determine the set of contiguous CLB columns (a stripe) required for all components of each level (call to `AddStripeInformation`). The final placement of a component will be restricted to the columns assigned to its level. The starting column assigned to a given

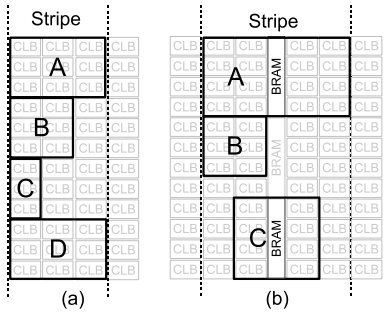


Figure 1. Placing components in stripes. (a) Typical placement for components that only have CLBs; (b) Placement resulting from restrictions imposed by the use of particular hardware resources, in this case BRAMs.

level will be the one closest to the dynamic area interface without overlapping columns of previous levels. The number of columns assigned to a stripe is the smallest required to accommodate all components of the corresponding level (see Figure 1a). This is determined by the width of the components and by the compatibility of the component resources with the destination area. In some cases it is necessary to widen the stripe in order to cover an area compatible with the resource requirements of a given component (see Figure 1b).

Placement proceeds by processing each level in succession and placing the components from top to bottom of the device. If possible, a new component is placed just below the previous one ($\texttt{YOffset}(c) = 0$) and at the right edge of the stripe ($\texttt{XOffset}(c) = 0$). However, the placement of components with non-homogeneous resources (like BRAMs) may require offsetting the component from the default location ($\texttt{YOffset}(c) > 0$ or $\texttt{YOffset}(c) > 0$). As a result, components may not start at the left edge of the stripe, nor end at the right edge. In all cases, the empty spaces in the stripe are filled with feed-through components, ensuring that all outputs are brought to the right side of stripe.

Feed-through components simply connect their inputs directly to their outputs. Components of this type are also used to provide a path through a stripe when connecting components that do not belong to the same level. The placer generates all feed-through components as required, without recourse to library components.

Placement fails if the sum of the heights of all components of the same level, including feed-through components added while processing previous levels, is greater than the height of the device. At the end of the placement procedure the information on the final positions of all component terminals is collected for use in the routing stage.

The automatic placement strategy assumes that components have input terminals on their left border and output terminals on the right. As an alternative to automatic placement, the run-time support library contains functions that allow the explicit placement of components by the application. In this case, both types of terminals may be present

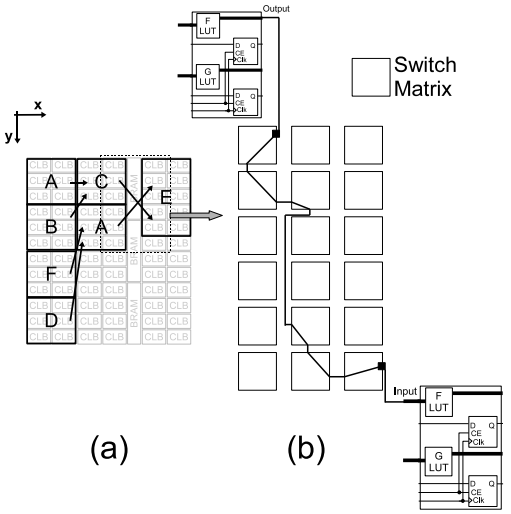


Figure 2. Routing between stripes. (a) Placed components with indication of connections to be established; (b) detail of routing area (dashed box) showing one possible route connecting C to E.

on either edge of the routing area. For both types of placement, a list of connections and associated physical terminal positions is created for use as input to the router.

The final result of a successful placement consists of the default partial bitstream merged with the relocated bitstreams of the components.

3.3. Routing

The routing procedure described in this section is used to establish connections between terminals of components in adjacent stripes. The procedure implements a breadth-first search of the routing area, which is represented by an array of switch matrices, one for each CLB in the area. For adjacent stripes, two columns of switch matrices are necessary: one belonging to the right border of the left stripe, and the other belonging to the left border of the right stripe. An extra column of switch matrices is included when there is an unused BRAM column between the stripes.

Physically, component terminals are pins of the switch matrix of the corresponding CLB. The component inputs connect to the input pins of the CLB LUTs, while component outputs connect to the slice outputs [18]. Other pins in the switch matrix connect to corresponding pins in other switch matrices. So the result of routing one net is simply the set of switch matrix pins required to establish the desired connectivity, which implicitly define the settings of the switch matrices involved. The situation is illustrated in Figure 2.

The actual area searched starts as the smallest rectangle of switch matrices that encloses all pins used as terminals of the net to be routed, and is reduced during the search. Since the search area is restricted, the number of possible connections to be examined is limited. Restricting the search area in this way reduces the chances of successfully routing a given netlist, but reduce the search effort signifi-

Algorithm 2: Greedy breadth-first routing algorithm

```
   Data: List R of nets to route
         Partial bitstream B with components
         Search area A
   Result: Partial bitstream B with merged routes
 1 usedPins ← create switch matrix array for search area A
 2 foreach n ∈ R do
 3     currentPins ← {GetNetSource(n)}
 4     destinationPins ← GetNetSinks(n)
 5     distLimit ← max(Width(A),Height(A))
 6     solutionPaths[n] ← ∅
 7     newCurrentPins ← ∅
 8     f ← SelectOne(destinationPins)
 9     while |Reached| < |destinationPins| do
10         if currentPins = ∅ then fail
11         allPins ← set of pins connected to any element of currentPins
12         foreach p ∈ allPins do
13             if p ∈ Visited then continue
14             Visited ← Visited ∩ {p}
15             d ← Distance(p, f)
16             if p ∉ usedPins ∧ InSearchArea(p) ∧ d ≤ distLimit
               then
17                 if p ∈ destinationPins then
18                     newPath ← RetracePathTo(p)
19                     MergePaths(solutionPaths[n], newPath)
20                     Reached ← Reached ∩ {p}
21                     if p = f then
22                         f ←
                           SelectOne(destinationPins \ Reached)
23                         Visited ← ∅
24                         newCurrentPins ←
                           {GetNetSource(n)}
25                         distLimit ←
                           max(Width(A), Height(A))
26                         break
27                 else
                       // it is not an endpoint
28                     newCurrentPins ← newCurrentPins ∩ {p}
29                     distLimit ← min(distLimit, d)
30         currentPins ← newCurrentPins
31     add all pins from paths in SolutionPaths[n] to usedPins
32     clear all flags, Visited ← ∅, Reached ← ∅
33 configure bitstream with all elements of solutionPaths[]
```

cantly. The restricted routing algorithm remains capable of routing significant classes of circuits, as shown empirically in section 4.

The high-level description of the routing algorithm for one region is shown in Algorithm 2. The algorithm performs a breadth-first search for a shortest-path forest between the source of a net (one component's output terminal) and its sinks (one or more input terminals). Nets are processed in sequence, without reconsidering the routing of previous nets (outer loop in line 2).

During the search, variable currentPins contains the pins that belong to the border of the expanding search, i.e., those pins that could be reached from the source in the number of steps corresponding to the number of iterations of the inner loop starting (line 9). Initially, only the source of the net belongs to this set (line 3); during execution, the successors of each visited pin are added (lines 28)

The loop at line 9 is repeated until every sink is reached. The distance of a pin p to the current target sink f is used to limit the search at line 16. The function `Distance(`p,f`)` (used in line 15) is equal to the largest of the vertical and horizontal distances between p and f: `Distance(`p,f`)` $= \max(|x_p - x_f|, |y_p - y_f|)$. Only pins within a distance distLimit are eligible for consideration. The value of distLimit is initialized to the largest dimension of the search area (generally, its height) and reduced as the search progresses (line 29). The variable is reset to the initial value after reaching each sink.

The variable newCurrentPins holds the set of pins to be used as starting points in the next iteration of the search. This set includes all pins directly connected to the pins in currentPins that have not yet been visited in the course of this search. Every pin added to newCurrentPins includes a reference to its predecessor on the search path. As the search is extended to neighboring pins, these are flagged as "visited", to avoid repeated processing and ensure that only shortest paths are considered.

Every time an element of destinationPins is reached, a path is created by retracing through the chain of predecessor pins (function `RetracePath()`). On reaching the current sink target, the search for any remaining endpoints of the same net is setup (line 21): newCurrentPins again contains only the source pin, Visited is now a empty set, and a new target sink is selected from the remaining ones. Once all sinks have been found (line 31), pins used in the solution are added to usedPins, state information for the current search is reset and the next net is processed. The final step updates the bitstream with the configuration information for the new routes.

The algorithm presented here does not ensure that a global optimum for all routes is obtained, since each net is treated in isolation, without considering the impact on the following nets. In addition, the dynamic restriction of the search area (for performance reasons) may cause some solutions to be ignored. The current implementation does not try to adjust the order in which nets are processed and does not control the congestion of the routing area during the search. The impact of these limitations is mitigated by the fact that the router's choices are considerably restricted by the previous placement, and by the design decision to keep any routing-related modifications confined to a relatively small inter-component area. As shown by the benchmark circuits of section 4, a large variety of circuits can be successfully routed by this approach.

4. Experimental results

The performance of the algorithms of section 3.3 was evaluated by applying them to a set of benchmark circuits. The evaluation was done on a XUP Virtex-II Pro Development System, which uses a Xilinx XC2VP30-7 FPGA [4] and 512 MB of external DDR memory (PC-3200). The external memory contains the program code and data, including the library of components. Only one of the two embedded PowerPC 405 processor cores is used for this work. The CPU operates at 300 MHz, and the 64-bit processor local bus connected to the memory controller employs a 100 MHz bus clock.

This section presents the results obtained by applying the placement and routing algorithms to three sets of benchmarks. For both sets, component dimensions and terminal positions have realistic values derived from actual designs.

The first set of benchmarks comprises four classes of

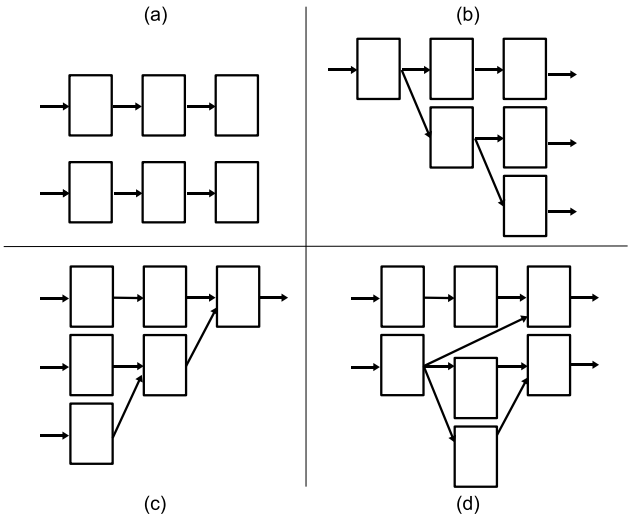


Figure 3. Example of circuit graphs for each class of the first set of benchmarks.

synthetic circuits, whose general structure is depicted in Figure 3:

Pipeline (a) One or more pipelines;

Tree SM (b) Tree-like graphs with a single input component and multiple output components;

Tree MS (c) Tree-like graphs with multiple input components and a single output component;

Random DAG (d) Random directed acyclic graphs.

The structure of the first three classes is well matched to the behavior of the placement algorithm, while the last class is more general.

Table 1 describes the basic characteristics of the individual examples: the number of input and output ports, the number of components working with each of the three different data sizes (8, 16 and 32 bits), the number of levels of the structure, and the maximum fan-out (number of sinks of a net).

The other two sets of benchmarks are an adaptation of benchmarks used by [19]:

Random binary expressions This set consists of 6 random binary expressions, which produce a binary tree structure, whose leaf nodes are the input constants and the root node is the result of the expression. All internal nodes are binary operations. The structural details of each benchmark are summarized in Table 2.

Honeywell/MediaBench The last set is based on nine data flow graphs adapted by [19] from the Honeywell [20] and MediaBench benchmarks [21]. All nodes are assumed to process 8-bit data items. Table 3 shows the structural details of all circuits from this set.

For the complete set of benchmarks, the average number of components is 15 and the average number of connections is 164.

	Number Inputs	Number Outputs	Levels	Number of Modules 8-bit	16-bit	32-bit	Nets	Maximum fan-out
Pipeline 1	8	8	3	3			32	1
Pipeline 2	16	16	3	6			64	1
Pipeline 3	24	24	4	12			120	1
Pipeline 4	24	24	4	4	4		120	1
Pipeline 5	32	32	4	4	4	1	160	1
Tree sm1	8	16	2	3			40	2
Tree sm2	8	32	3	7			88	2
Tree sm3	16	64	3	4	5		176	2
Tree sm4	32	64	4	12	2	3	288	2
Tree sm5	32	64	4	16	4	1	288	4
Tree ms1	32	8	2		3		48	1
Tree ms2	32	16	3	3			128	1
Tree ms3	64	32	3	4	5	1	224	1
Tree ms4	64	32	4	12	2	1	288	1
Tree ms5	64	8	4	12	8	1	320	1
Random DAG 1	16	8	3	5			72	2
Random DAG 2	32	32	3	5	2		112	2
Random DAG 3	32	32	4	7	5		208	2
Random DAG 4	32	32	5	5	6	1	264	2
Random DAG 5	32	32	5	7	4	2	288	4

Table 1. Basic structural characteristics of all example circuits from the first set of synthetic benchmarks.

	Number Inputs	Number Outputs	Levels	Number of 8-bit Modules	Nets
tg01	40	8	4	9	72
tg02	80	8	5	19	152
tg03	48	8	4	11	88
tg04	80	8	6	19	152
tg05	80	8	5	19	152
tg06	120	8	5	25	200

Table 2. Basic structural characteristics of the second set of benchmarks representing binary random expressions. The names of the benchmarks are the ones used in [19].

The program used to run the benchmarks was written in C and compiled with the GNU Compiler version 3.4.1 included in EDK 8.2. The resulting programs has 105 KB of instructions and 1597 KB of static data.

Table 4 summarizes the results of running the place and route algorithms on the benchmark circuits. For each benchmark, Table 4 presents the total time required for bitstream generation, the number of levels of the corresponding graph, the smallest rectangular area occupied by the resulting circuit, the number of feed-through CLBs added during routing, the number of CLBs taken up by all components, including those used for feed-through routing. The last column shows the relative area occupied by feed-

	Number Inputs	Number Outputs	Levels	Number of 8-bit Modules	Nets
Honeywell-intfc01	24	40	4	13	104
Dft	56	32	3	12	64
Honeywell-versatil01	24	40	5	20	144
Honeywell-intfc02	48	56	7	21	152
Honeywell-fft01	40	64	6	23	168
Honeywell-fft02	56	56	7	24	184
MediaBench-jpeg	56	128	5	27	200
Honeywell-fft03	48	32	8	28	240
Honeywell-versatil02	72	104	8	41	328

Table 3. Basic structural characteristics of the circuits from the Honeywell and MediaBench benchmarks (adapted from [19]).

	Time (s)	Bounding box (Columns x Rows)	Number of feedthroughs	Total component area (CLBs)	Area used for feedthoughs (%)
Pipeline 1	6.97	6x3	0	18	0
Pipeline 2	13.67	6x6	0	36	0
Pipeline 3	23.94	8x12	0	96	0
Pipeline 4	26.63	12x9	4	96	4
Pipeline 5	39.11	12x12	4	132	3
Tree sm1	9.39	6x6	1	24	4
Tree sm2	20.92	8x12	4	66	6
Tree sm3	45.23	9x24	4	132	3
Tree sm4	92.85	11x24	4	180	2
Tree sm5	99.73	12x24	12	192	6
Tree ms1	10.73	5x8	0	30	0
Tree ms2	27.14	9x12	2	90	2
Tree ms3	73.54	9x24	4	144	3
Tree ms4	94.17	12x24	8	216	4
Tree ms5	96.90	12x32	16	256	6
Random DAG 1	16.28	8x6	2	46	4
Random DAG 2	26.21	9x16	6	102	6
Random DAG 3	50.35	12x24	12	156	7
Random DAG 4	86.44	16x32	22	185	11
Random DAG 5	92.43	24x32	30	200	13
tg01	18.84	10x11	0	71	0
tg02	57.78	13x17	0	151	0
tg03	22.03	10x18	0	96	0
tg04	54.87	16x17	0	166	0
tg05	55.69	13x17	0	151	0
tg06	66.94	13x27	0	211	0
Honeywell-intfc01	20.86	21x16	18	156	10
Dft	13.62	9x28	0	144	0
Honeywell-versatil01	40.58	18x29	24	240	9
Honeywell-intfc02	44.14	21x29	15	252	6
Honeywell-fft01	45.62	18x33	6	276	2
Honeywell-fft02	62.91	21x32	24	288	8
MediaBench-jpeg	53.16	15x32	18	324	5
Honeywell-fft03	81.67	24x35	36	336	10
Honeywell-versatil02	100.73	24x56	15	492	3

Table 4. Results of the execution of the placement and routing algorithms on the 300 MHz PowerPC 405 embedded in the XC2VP30-7 FPGA.

through components.

The running time is completely determined by the routing stage. The most time-consuming placement took only 154 ms for the Honeywell-fft03 benchmark. The number of levels L is equal to the number of stripes. Therefore, the routing procedure (Algorithm 2) is called $L+1$ times for each benchmark, for connections between strips and connections for the input and outputs. For Virtex-II-pro FPGAs the size of the partial bitstream, and therefore the time taking for partial reconfiguration, is proportional to the number of columns occupied by the circuit (first number in the fourth column). The typical dynamic area of our test system is 22 columns by 32 rows. Most of the benchmarks fit this reserved area; the four that do not, still fit comfortably our target FPGA, which has 46 columns by 80 rows.

Routing may involve adding feed-through components to the circuit in order to connect components that are not on successive levels. With the exception of two benchmarks (the two largest random DAGs), the additional components do not represent more than 10% of the total number of CLBs used by all components.

Most benchmarks took less than 90 s; the exceptions are the two of the largest trees (of both types), the largest random DAG and the largest Honeywell benchmark. The global average running time is 48.3 s. These running times make the current version unsuitable for applications that require a very fast turnaround time, like just-in-time compilation.

For the hardware setup used in this evaluation, a one-time reduction in running time might be obtained by using both CPU cores: since the routing area between stripes can be processed independently, routing may be easily performed concurrently by both processors. Another possibility, applicable to situations in which partial configurations are reused during the same application run, is to maintain a configuration cache.

There are, in addition, many application scenarios that may accommodate delays in the range under discussion. They include applications that must adapt to relatively slow-changing environments (like exterior lighting conditions or temperature) or that may operate temporarily with reduced quality. Another scenario involves adaptive systems that use learning (for instance, of new filter settings) to improve their performance: the time required for generating configurations may be only a part of the time necessary to learn the new settings and to take the decision to switch configurations.

Another application involves self-diagnosis of malfunctioning systems. In this case, normal operation has not yet begun (or has been interrupted). Depending on the results of some initial self-tests, the system may proceed to a diagnosis phase, during which new test hardware is generated which depends on the results of the previous tests. In this case, run-time generation would avoid the need to pre-generate and store a potentially very large number of specific diagnostic circuits (most of which would never be used).

The current system is also useful in adapting components to a design-specific dynamic area interface. Often, it is desirable to re-use some (large) component in several systems having different configurations of the dynamic area (in particular, the position of the connections between the dynamic and static areas may change). The component might even be a third-party intellectual property block, designed without any knowledge of the physical details of the dynamic area. With the current system, the physical interface adaptation might be performed at run-time by routing the appropriate connections between the reserved area interface and the component.

5. Conclusion

This paper presents the first implementation and evaluation of an embedded system that is able to generate partial bitstreams at run-time for use in the dynamic reconfiguration of sections of a Virtex-II Pro platform FPGA. The goal is to obtain useful solutions in a short time. The system uses a greedy placement heuristic based on topological sorting to determine the positions of individual coarse-grained components whose interconnections are specified by an acyclic netlist. A router based on non-backtracking search over restricted areas determines the routes for the interconnections. The partial bitstream is constructed by merging together a default bitstream of the reconfigurable area, the relocated partial bitstreams of the components, and the configuration of the switch matrices used for routing. The computational effort is kept bounded by a combination of factors: circuit description by acyclic netlists of coarse-grained components, simplified resource models,

direct placement procedure, and the restricting of routing to limited areas.

The results for a set of 35 benchmarks (both synthetic and application-derived) show that time required for bitstream generation on a 300 MHz PowerPC embedded processor depends strongly on the complexity of the circuits, averaging 48.3 s (minimum: 6.97 s, maximum: 100.73 s) for an average circuit size of 15 components (minimum: 3, maximum: 41) and 164 connections (minimum: 32, maximum: 328).

The working implementation described here shows that run-time generation of configurations is a feasible technique for use on highly adaptive embedded systems, where it may be used to provide precisely-tailored hardware support to tasks whose computational needs exceed the computational power of the CPU. The evaluation of the suitability of this approach for specific cases requires that all system aspects be considered. Although the time required for routing makes the approach unsuitable for applications requiring very fast generation of bitstreams, several classes of applications may be able to accommodate the delays involved and profit from the increased flexibility provided by this approach.

Acknowledgments

The authors would like to thank C. Ababei for providing some of the benchmarks used in section 4.

References

[1] J. Gause, P.Y.K. Cheung, and W. Luk. Reconfigurable computing for shape-adaptive video processing. *IEE Proceedings - Computers and Digital Techniques*, 151(5):313–320, 2004.

[2] K. Paulsson, M. Hiibner, J. Becker, J.-M. Philippe, and C. Gamrat. On-line routing of reconfigurable functions for future self-adaptive systems - investigations within the ÆTHER project. In *International Conference on Field Programmable Logic and Applications (FPL 2007)*, pages 415–422, 2007.

[3] Miguel L. Silva and João C. Ferreira. Generation of hardware modules for run-time reconfigurable hybrid CPU/FPGA systems. *IET Computers & Digital Techniques*, 1(5):461–471, 2007.

[4] Xilinx. *Virtex-II Platform FPGA User Guide*, November 2007. version 2.2.

[5] Ian Robertson and James Irvine. A design flow for partially reconfigurable hardware. *ACM Transactions on Embedded Computing Systems*, 3(2):257–283, 2004.

[6] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford. Invited paper: Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs. In *Proc. International Conference on Field Programmable Logic and Applications (FPL 2006)*, pages 1–6, 2006.

[7] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour. Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In *Proc. 39th Design Automation Conference*, pages 343–348, 2002.

[8] Y.E. Krasteva, E. de la Torre, T. Riesgo, and D. Joly. Virtex II FPGA bitstream manipulation: Application to reconfiguration control systems. In *Proc. International Conference on Field Programmable Logic and Applications (FPL 2006)*, pages 1–4, 2006.

[9] Heiko Kalte and Mario Porrmann. REPLICA2Pro: Task relocation by bitstream manipulation in Virtex-II/Pro FPGAs. In *Proceedings of the 3rd Conference on Computing Frontiers*, pages 403–412. ACM, 2006.

[10] F. Ferrandi, M. Morandi, M. Novati, M. D. Santambrogio, and D. Sciuto. Dynamic reconfiguration: Core relocation via partial bitstreams filtering with minimal overhead. In *Proc. International Symposium on System-on-Chip (Soc 2006)*, pages 1–4, 2006.

[11] H. Tan and R. F. DeMara. A multilayer framework supporting autonomous run-time partial reconfiguration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(5):504–516, 2008.

[12] Yaska Sankar and Jonathan Rose. Trading quality for compile time: ultra-fast placement for FPGAs. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, pages 157–166. ACM, 1999.

[13] Chandra Mulpuri and Scott Hauck. Runtime and quality tradeoffs in FPGA placement and routing. In *Proceedings of the 2001 ACM/SIGDA 9th International Symposium on Field-Programmable Gate Arrays*, pages 29–36. ACM, 2001.

[14] Roman Lysecky, Frank Vahid, and Sheldon X.-D. Tan. Dynamic FPGA routing for just-in-time FPGA compilation. In *Proc. 41st Design Automation Conference*, pages 954–959, 2004.

[15] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[16] Jorge Suris, Cameron Patterson, and Peter Athanas. An efficient run-time router for connecting modules in FPGAS. In *Proc. International Conference on Field Programmable Logic and Applications (FPL 2008)*, pages 125–130, 2008.

[17] Miguel L. Silva and João C. Ferreira. Generation of partial FPGA configurations at run-time. In *Proc. International Conference on Field Programmable Logic and Applications (FPL 2008)*, pages 367–372, 2008.

[18] Xilinx. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, November 2007. version 4.7.

[19] Cristinel Ababei and Kia Bazargan. Non-contiguous linear placement for reconfigurable fabrics. *International Journal of Embedded Systems*, 2(1/2):86–94, 2006.

[20] S. Kumar, L. Pires, S. Ponnuswamy, C. Nanavati, J. Golusky, M. Vojta, S. Wadi, D. Pandalai, and H. Spaanenberg. A benchmark suite for evaluating configurable computing systems—status, reflections, and future directions. In *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays*, pages 126–134. ACM, 2000.

[21] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *In International Symposium on Microarchitecture*, pages 330–335, 1997.

Interligação intra- e inter-circuito de componentes especificados com Redes de Petri

Ricardo Ferreira[1] Anikó Costa[1,2] Luís Gomes[1,2]
ricardowf@fct.unl.pt *akc@uninova.pt* *lugo@fct.unl.pt*

[1] *Universidade Nova de Lisboa – Faculdade de Ciências e Tecnologia*
[2] *UNINOVA – CTS*

Sumário

Este trabalho tem como objectivo a apresentação de um ambiente de desenvolvimento que permita a geração automática de código para a interligação de componentes obtidos como resultado da partição de um modelo expresso em redes de Petri em diferentes plataformas. A solução de interligação proposta recorre a uma solução Network-on-Chip com suporte a comunicações intra- e inter-circuito, baseada no protocolo RS-232 (embora possa funcionar a ritmos de transmissão mais elevados). O resultado obtido será implementado em plataformas reconfiguráveis da Xilinx, FPGA Spartan3 e Virtex-II, e em microcontroladores de baixo custo, PIC 18F4620 da Microchip.

Considerando uma topologia em anel e utilizando o protocolo série RS-232 (suportado pela generalidade dos microcontroladores) serão apresentadas as regras e considerações necessárias para que seja possível gerar este tipo de soluções. Finalmente, será apresentado um exemplo onde, através de um modelo expresso em redes de Petri, se apresentará o fluxo de desenvolvimento e como são aplicadas as regras apresentadas.

1. Introdução

A evolução no hardware tem vindo a possibilitar a integração de múltiplos componentes num único chip, tais como processadores, controladores dedicados e memórias, resultando na integração de um sistema completo num mesmo integrado. Este tipo de solução poderá recorrer a um circuito dedicado ou a um dispositivo reconfigurável. Estes sistemas são, normalmente, denominados *Systems-on-Chip* (SoCs) ou *Systems-on-a-Programmable-Chip* (SoPCs). É comum o abuso de linguagem referindo como SoC as soluções SoPC.

Este aumento da complexidade dos sistemas faz com que as exigências de modelação sejam maiores e que seja comum encarar a divisão do sistema em vários subsistemas interactuantes. Do ponto de vista da comunicação entre estes subsistemas, o uso de ligações dedicadas é, normalmente, inviável, pois, apesar de oferecer melhor desempenho, ocupa demasiada área do ponto de vista da sua implementação SoC/SoPC ou necessitaria de cablagem específica quando se considerassem subsistemas implementados de forma heterogénea. Justifica-se, desta forma, a utilização de soluções de interligação dos subsistemas através de uma rede dedicada, dando origem às normalmente designadas NoCs (Network-on-Chip).

Tal como referido em [1], a ideia fundamental numa Network-on-Chip é aplicar a abordagem de camadas comum em sistemas de telecomunicações e em redes de computadores. O modelo de referência OSI (Open Systems Initiative) não é, necessariamente, seguido com rigor, sendo, normalmente, adaptado. A rede pode ser orientada à ligação (comutação de circuitos) ou não orientada à ligação (comutação de pacotes). É sempre necessária uma interface própria para formar o pacote e/ou estabelecer a ligação. A NoC é composta por nós (também designados comutadores) que encaminham o tráfego, podendo conter *buffers*. A topologia utilizada influência a forma como os recursos da rede se interligarão. As propostas de topologias existentes são variadas, incluindo, topologias de interligação ad-hoc, em malha (*mesh*), em malha toroidal, em anel bidireccional, octogonais ou em árvore [1].

Em [1] apresentam-se algumas das propostas mais comuns existentes para NoCs, das quais se realçam as seguintes:

a) ***xPipes*** [2] – uma rede flexível constituída pela parametrização de componentes sintetizáveis. A topologia pode ser especificada pelo projectista. Recorre-se a comutação com tabelas de encaminhamento estáticas. A rede é síncrona, e quer os comutadores, quer as ligações, são *pipelined* de modo a obter uma elevada taxa de transmissão. Um pacote de confirmação de recepção é devolvido ao transmissor após sucesso da respectiva transmissão;

b) ***Proteo*** [3] – é semelhante à *xPipes* em muitos aspectos. As implementações são baseadas em interligações parametrizadas de blocos *IP* e a topologia pode ser seleccionada pelo projectista. Existe a opção de utilização de pacotes de confirmação, mas a implementação da retransmissão do pacote ou a correcção de erros estão a cargo do projectista;

c) ***Nostrum*** [4] – utiliza a topologia "clássica" em malha, com os recursos colocados nos núcleos ligados por uma matriz de interligação. O modelo OSI é utilizado, mas só as 3 camadas mais baixas (física, lógica e camada de rede) são obrigatórias. É possível formar um circuito virtual de forma a que uma fracção do total da largura de banda disponível possa ser alocada a serviços prioritários, (implementado com base numa estratégia *Time-Division Multiple Access* (TDMA) modificada);

d) ***SoCbus*** [5] – *SoCbus* (NoC circuito-comutado) utiliza uma topologia em malha 2D. O objectivo é substituir os barramentos embutidos por uma rede de comutação de circuitos para fornecer maior largura de banda. Há um único nó de coordenação - *Central Coordination Node* (CNN) - executando as funções de coordenação do sistema. O CNN gera a configuração de cada nó quando um circuito é inicializado. Não existe controle de fluxo garantido, mas existe um sinal de confirmação;

e) ***SPIN fat tree*** [6] – Ao contrário de outras NoCs, a *Scalable Programmable Integrated Network* (SPIN) tem como topologia de referência uma "*fat-tree*". Trata-se de uma estrutura em árvore com encaminhadores nos nós e recursos computacionais nas folhas, excepto nos nós com "pais" repetidos. Não considera detecções de erros ou retransmissões;

f) ***XGFT*** [7] – As *Generalized Extended Fat Trees* (XGFTs) podem ser construídas com base em nós que encaminham os pacotes no sentido ascendente e no sentido descendente com blocos de comutação XGFT separados; foi provado que o desempenho da solução XGFT é superior ao das soluções em malha;

g) ***Redes CDMA*** [8] – O *Code-Division Multiple Access* (CDMA) é uma técnica bem conhecida utilizada nas comunicações sem fios spread-spectrum. Também foi aplicado a barramentos embutidos. É necessário um árbitro centralizado para configurar a interface de destino de forma a receber o código do canal desejado; e

h) ***Philips Æthereal*** [9] – utiliza encaminhamento sem contenção, ou *pipelined time-division-multiplexed circuit switching*, para implementar os seus serviços de desempenho garantido. Embora todas as *streams* de dados tenham a mesma prioridade, podem utilizar reservas de largura de banda diferentes. Os *slots* de tempo e o encaminhamento são "programados" usando tabelas residentes em cada nó. Há dois modelos de programação: distribuídos e centralizados.

Por outro lado, as metodologias actuais utilizadas no desenvolvimento de sistemas electrónicos integrando componentes hardware e software têm vindo a dar relevância crescente aos modelos (tendência para metodologias baseadas em modelos, MBD – *Model-based Development*). Ter um modelo que descreva adequadamente o sistema é uma mais-valia para o seu desenvolvimento e para a sua documentação. Se a partir do modelo se gerar código automaticamente poupar-se-á tempo gasto no desenvolvimento e, eventualmente, evitar-se-ão erros de implementação manual. A adopção de abordagens de desenvolvimento baseadas em modelos pode suportar melhorias claras no fluxo de desenvolvimento de sistemas, como referido em [10] e [11].

O projecto FORDESIGN [12], concluído recentemente, enquadra-se nestes objectivos, recorrendo à utilização de modelos expressos em redes de Petri (RdP) para a especificação do comportamento do sistema. No referido projecto foi desenvolvida uma classe de Redes de Petri denominada por IOPT – Input-Output Place-Transition Petri nets [13], permitindo integrar dependências em termos de sinais e eventos de entrada e de saída, suportando a modelação adequada de controladores. No conjunto de ferramentas desenvolvido inclui-se uma ferramenta de partição de um modelo IOPT em vários submodelos, posteriormente considerados como componentes em execução paralela (do ponto de vista da implementação).

Os componentes referidos serão obtidos como resultado da partição de um modelo expresso em redes de Petri, de acordo com as regras de partição propostas em [14], realizada utilizando o editor de RdP SNOOPY-IOPT [12] em conjugação com a ferramenta SPLIT [12] e com as ferramentas de geração automática de código C e VHDL a partir das representações PNML dos modelos RdP resultantes da partição.

O objectivo principal deste trabalho é o de apresentar uma solução desenvolvida para a interligação dos componentes gerados, recorrendo a soluções do tipo Network-on-Chip (NoC).

A solução de interligação será codificada em VHDL nas plataformas de implementação previstas para a validação da solução (onde se incluem as FPGAs da Xilinx das famílias Spartan-3 e Virtex-II), bem como em C para os sistemas externos com capacidade para suportar a solução de interligação pretendida (nomeadamente computadores de utilização geral e microcontroladores de baixo custo, nomeadamente os PIC da Microchip).

Para que se possa garantir a interligação de diferentes plataformas não foi considerada uma

Network-on-Chip comum, mas sim uma *Network-on-Chip* que permita suportar comunicações intra- e inter-circuito, baseada no protocolo série RS-232 (podendo, embora, operar a velocidades muito superiores). Desta forma, incluem-se as soluções de hardware reconfigurável, os microcontroladores de baixo-custo comummente utilizados em tarefas de controlo, bem como a interligação a outros sistemas computacionais possuidores de interfaces normalizados do tipo RS-232. A selecção de um suporte à comunicação com as características do RS-232 tem várias vantagens, de entre as quais se realçam: (1) estar disponível na maior parte dos dispositivos controladores; (2) permitir uma fácil ligação; (3) ser de baixo custo; e (4) permitir a integração com sistemas distribuídos de automação, nomeadamente com os equipamentos associados, como os sistemas de manufactura e os de controlo predial. Como desvantagem principal deve ser referida o ritmo de transferência de dados relativamente baixo, quando comparado com outras soluções. No entanto, considera-se que a flexibilidade referida permitindo interligar dispositivos intra-circuito e inter-circuitos compensa as limitações referidas.

Desta forma, o objectivo é o de obter uma solução (e uma ferramenta que permita automatizar a sua geração) que, embora com um desempenho com algumas limitações, possua uma elevada flexibilidade baseada numa rede de comunicação série e que permita gerar o código (VHDL e/ou C) da solução de interligação, partindo dos componentes associados expressos através dos seus modelos RdP-IOPT.

Esta comunicação encontra-se dividida em várias secções, seguindo-se a secção 2 onde se procede a uma descrição geral do sistema, na secção 3 se apresenta um exemplo de aplicação onde várias topologias para a interligação dos componentes são apresentadas e comparadas, e a secção 4 onde se conclui e se apresentam os trabalhos futuros.

2. Descrição do sistema

Tal como referido, o objectivo deste trabalho é o de gerar, de forma automática, o código de uma solução de interligação de vários módulos obtidos anteriormente através da ferramenta de partição de modelos RdP SPLIT e das ferramentas de geração automática de código denominadas PNML2C e PNML2VHDL, em que código C e VHDL é obtido a partir de ficheiros representando modelos RdP-IOPT expressos em PNML (Petri Nets Markup Language) [15].

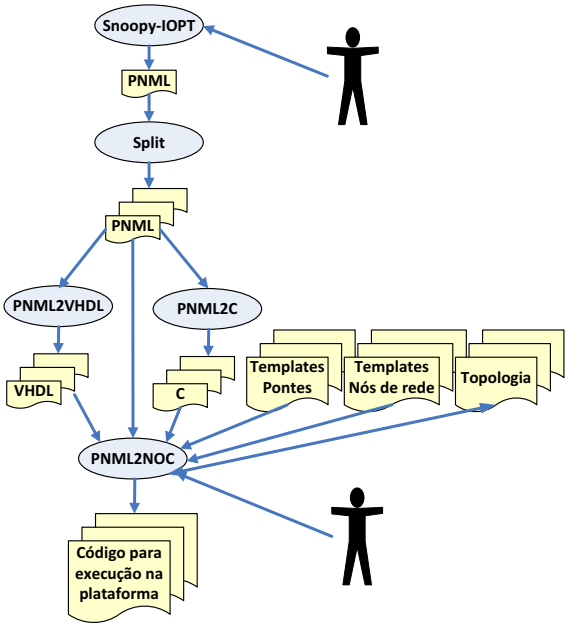


Figura 1. Fluxo de desenvolvimento e ferramentas de suporte.

Para o seu desenvolvimento foram consideradas as ferramentas atrás referidas, desenvolvidas no âmbito do projecto FORDESIGN (Figura 1). O SNOOPY-IOPT permite a edição de redes de Petri IOPT e posterior representação num ficheiro PNML. As ferramentas PNML2C e PNML2VHDL geram automaticamente o código C e VHDL, respectivamente, a partir dos ficheiros PNML obtidos pela ferramenta SPLIT.

A selecção do PNML2C e/ou do PNML2VHDL para a geração do código associado a um determinado componente está relacionada com as plataformas escolhidas para a execução desses componentes; os *templates* que se irão utilizar para produzir os nós de interligação através da NoC também são afectados por essa escolha. Estes últimos são ficheiros VHDL e C que serão alterados conforme as definições de rede. Estas definições estão num ficheiro (utilizando formato XML) e ditam qual a topologia, a posição de cada componente e velocidades na rede.

É utilizado o protocolo lógico RS-232 com uma topologia em duplo anel (duplo *daisy-chain*) para garantir a interligação, onde cada módulo possui um componente (associado a um submodelo gerado pela partição da RdP) e um nó de rede, através do qual se ligará à rede.

Cada nó tem o seu próprio endereço de rede, bem como a sua posição na topologia, que são definidos tendo em conta o número de mensagem enviadas/recebidas, o número de saltos e o ritmo de transmissão da rede.

O número de mensagens é retirado dos ficheiros PNML que caracterizam cada submodelo tendo em

conta o seu número de sinais de entrada e saída, enquanto que a velocidade de transmissão é previamente definida conforme a plataforma onde o sistema for instalado. O número de saltos é calculado em função do endereço escolhido, pois é com base neste endereço que é determinada a posição do nó na topologia.

Cada mensagem contém o endereço do próprio nó, do nó a quem se destina a mensagem, a designação do sinal contido e o seu valor (Figura 2). Como, tipicamente, as RdP-IOPT deste trabalho são redes que modelam controladores, é necessário assegurar a qualidade de serviço, respondendo com uma mensagem de *acknowledge* ou de *not acknowledge*. Dependendo do código de resposta, o transmissor poderá ter que reenviar a mensagem ou simplesmente ficar prevenido que o destinatário tem o seu *buffer* cheio e até receber uma mensagem de reconhecimento não lhe enviará mais nada. No pior cenário, onde não chega qualquer mensagem de confirmação (porque ocorreu um erro na rede), expirará um temporizador, calculado tendo em conta a velocidade de transmissão da rede, o tempo de processamento de cada nó e a sua capacidade de armazenamento.

Figura 2. Formatos das mensagens a) de envio, b) de *acknowledge* e c) de *not acknowledge*

Tal como foi referido anteriormente, a transmissão é feita com base no protocolo lógico já utilizado na norma RS-232 (embora utilizando um ritmo de transmissão muito superior, quando realizado intra-circuito). Desta forma, as mensagens a transmitir irão ser compostas por múltiplos de oito bits, mais um de paridade, pelo que terão sempre um tamanho constante múltiplo de 11 bits (considerando *start* e *stop* bits). Este tamanho será determinado pelo número de bits necessários para a codificação dos endereços dos nós envolvidos, e de qualquer dos sinais (dependendo do tipo existente com o maior tamanho e de quantos sinais existem). O bit inicial da mensagem a transmitir indicará se a trama a transmitir é, ou não, superior a 8 bits (camada física do modelo OSI), permitindo distinguir entre o primeiro byte e os restantes bytes de uma mensagem

A topologia utilizada para a circulação de mensagens é composta por dois anéis. Um no sentido horário, outro no sentido anti-horário, tentando evitar que, caso uma das ligações se parta ou fique "empastelada", a rede entre num estado de bloqueio. Cada nó tem uma tabela de encaminhamento, de forma a optimizar o envio das mensagens, escolhendo um dos dois anéis de comunicação.

Na figura 3 é apresentada a estrutura do nó de rede onde se destacam dois módulos. Um para o processamento da informação que chega e sai do módulo e outro que gere a ligação de sinais e eventos, quer de entrada quer de saída, com o código de execução associado ao modelo RdP-IOPT (este último obtido directamente através do gerador de código automático). Existem também quatro *buffers* para recepção e envio das tramas RS232 e outros dois para os sinais e/ou eventos de interligação com o código de execução do modelo. O comportamento do nó poderá ser descrito como:

1. Depois de recebidas os bytes necessários para se ter qualquer uma das mensagens atrás referidas (figura 2), a trama recebida será retirada do *buffer*;
2. Será analisado se o endereço de destino coincide com o endereço do nó. Se não forem iguais é sinal que a mensagem não era destinada a este nó mas sim a outro, caso em que a mesma mensagem é colocada no buffer de saída (no mesmo sentido);
3. Caso a mensagem seja realmente destinada ao nó de rede, os sinais, eventos e respectivos valores são, depois de retirados da mensagem, colocados num buffer específico para sinais de entrada do modelo. É também criada uma mensagem ACK que será devolvida à origem;
4. Uma vez executado o modelo, as suas saídas serão colocadas num *buffer* semelhante ao anterior mas para sinais de saída. Sempre que este *buffer* não esteja vazio, será analisada a tabela de encaminhamento de forma a saber qual o destino do sinal. É composta uma mensagem e, conforme o destino, é decidido em qual dos dois *buffers* RS-232 de saída (sentido horário ou anti-horário) será colocada.
5. Sempre que o buffer esteja cheio, ou que uma mensagem seja ilegal, é criada uma mensagem NACK e colocada no buffer RS232 de saída de forma a ser entregue ao emissor.

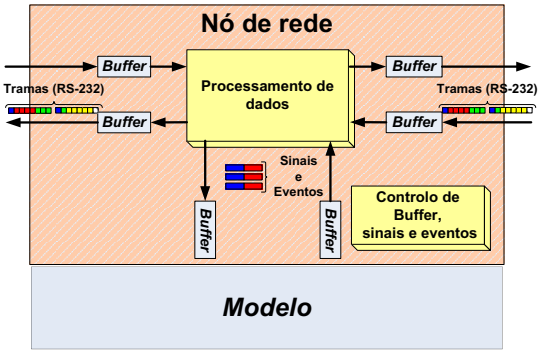


Figura 3. Nó de rede

978-972-789-304-1 REC'2010

Quando o sistema incluir mais do que uma plataforma de implementação, existe um nó de rede diferente que tomará a designação de ponte. A ponte é semelhante ao nó de rede, no entanto não tem a ligação ao modelo inicialmente caracterizado e sim mais uma ligação (TX e RX) RS-232. De acordo com a figura 3, tem apenas o processamento de informação e os buffers RS-232. A ponte ficará, tipicamente, na plataforma mais rápida, pois é através dela que conseguimos balancear a velocidade de transmissão dentro e fora da plataforma. Assim do outro lado (a outra plataforma) ter-se-á um nó cuja única diferença é não estar ligado em anel e ser responsável pelo controlo dos sinais e/ou eventos tal como referido atrás (figura 4).

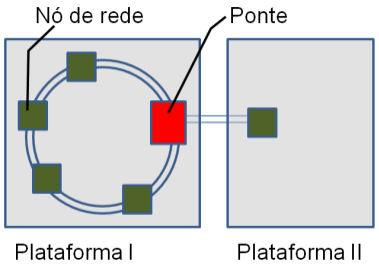


Figura 4. Arquitectura de ligação entre nós em plataformas distintas utilizando uma ponte.

3. Exemplo de aplicação

Como exemplo de aplicação de um sistema de automação em que é benéfica a execução concorrente de submodelos resultantes da partição do modelo, vamos recorrer ao exemplo inicialmente introduzido em [16], onde o sistema a controlar é composto por três carros que se movimentam entre dois pontos A_i e B_i sincronizando-se o seu movimento através dos botões de *GO* e *BACK*. O sistema está ilustrado na **Figura 5**.

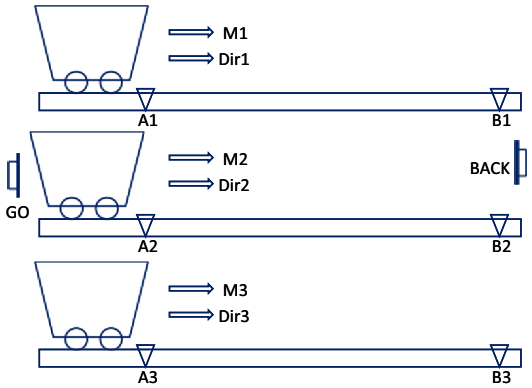


Figura 5. Exemplo de aplicação.

Sendo assim, o controlador do sistema tem como sinais de entrada A1, A2, A3 sinalizando os pontos de início de cada trajectória, B1, B2 e B3 instalados no fim do percurso, e os botões de sincronização dos movimentos GOe BACK, dando ordem para os carros iniciem o seu movimento em direcção de B ou A respectivamente. Como sinais de saída consideramos M1, M2, M3 indicando que o carro está em movimento e Dir1, Dir2, Dir3 indicando a direcção do movimento.

Considerando as redes de Petri IOPT (*Input-Output Place-Transition*) [13] como formalismo de modelação para o comportamento do controlador do sistema obtemos a rede apresentado na **Figura 6**. Para não sobrecarregar a figura, os sinais de saída associados à actuação dos motores, apesar de definidos, não estão visíveis no modelo.

Tendo em conta que o nosso objectivo é obter um controlador para cada carro temos de decompor este modelo em três submodelos que podem ser considerados como o modelo do controlador de cada um dos carros. Para esse efeito vamos utilizar a operação *Net Splitting* [14] e a ferramenta de SPLIT. Na **Figura 6** estão assinalados os nós por onde se deve partir o modelo para obter a decomposição desejada. Considerações sobre a selecção do conjunto de nós para o conjunto de corte do modelo inicial ficam fora do âmbito desta comunicação; no entanto, importa referir que, de uma forma geral, a sua selecção é realizada tendo em conta o modelo inicial e um objectivo específico (neste caso obter um controlador para cada um dos carros do sistema).

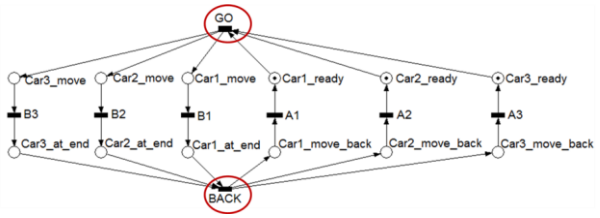


Figura 6. Modelo RdP do controlador do sistema, com o conjunto de corte assinalado.

Tendo em conta que o formalismo de modelação utilizado é o das redes de Petri IOPT serão incluídos os eventos internos gerados nos modelos resultantes da partição do modelo. Estes eventos gerados são associados aos canais de comunicação. Na interligação entre os submodelos considera-se a interligação dos eventos de saída e entrada com nomes semelhantes, como ilustrado na **Figura 7**, onde os eventos de nome "inevent???" e "outevent???" garantem a comunicação entre as transições "???_master" e "???_slave".

A ferramenta SPLIT, que executa a decomposição do modelo, gera estes eventos de comunicação de forma a permitir identificação automática para efeitos de interligação. Na **Figura 7** estão representados os módulos que vão ser implementados em cada controlador. Neste modelo estão incluídos os eventos de entrada e saída gerados, bem como a sua interligação e os sinais de entrada e saída que comunicam com o mundo exterior.

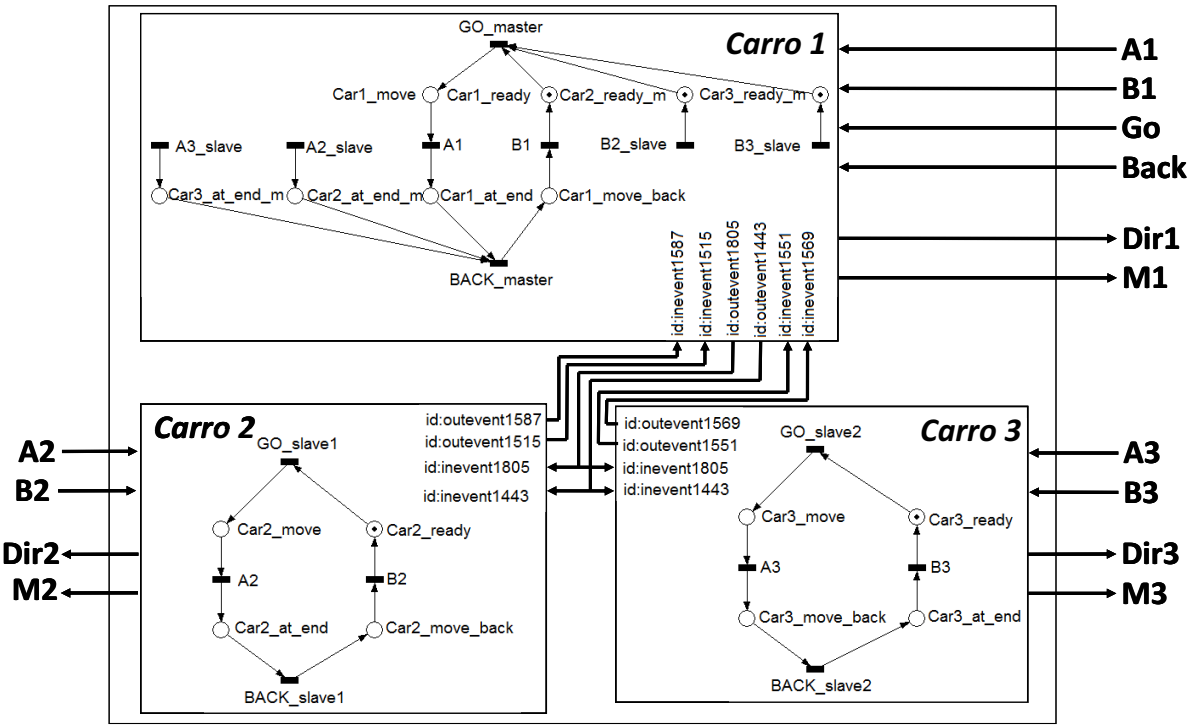


Figura 7. Modelos dos controladores dos vários caros, resultantes da operação SPLIT, a) carro1, b) carro2 e c) carro3.

A ferramenta SPLIT lê o ficheiro PNML contendo o modelo, bem como o conjunto de corte a utilizar na partição do modelo, e gera os submodelos e os ficheiros PNML associados para cada módulo.

Uma vez obtidos os ficheiros PNML de cada submodelo, são analisados os sinais e/ou eventos de cada um dos submodelos de forma a ser construída a tabela que nos permitirá definir o endereço e a topologia da rede de interligação (tabela 1).

Tabela 1. Análise do número de mensagens e ritmo de transmissão entre modelos.

Número de sinais/eventos

Origem \ Destino	Carro1	Carro2	Carro3
Carro1	-	2	2
Carro2	2	-	0
Carro3	2	0	-

Ritmo de Transmissão [bps]

Origem \ Destino	Carro1	Carro2	Carro3
Carro1	-	9600	9600
Carro2	9600	-	9600
Carro3	9600	9600	-

Da análise das tabelas, conclui-se que o carro1 é o componente que é responsável pela geração de mais mensagens e como tal deve ficar entre os outros dois modelos. Assim sendo, é atribuído o endereço de rede #00 ao carro2, #01 ao carro1 e #10 para o carro3.

Uma vez definidos os endereços de cada nó, é necessário calcular a tabela de encaminhamento para cada um. Para tal voltam a ser analisados os sinais e eventos dos ficheiros PNML. A tabela 2 mostra como será a tabela de encaminhamento do carro 1.

Tabela 2. Tabela de encaminhamento do nó de rede do modelo carro1.

Sinal	*# bits*	*Destino*	*Sentido*
outevent1805	1	#00	horário
outevent1805	1	#10	anti-horário
outevent1443	1	#00	horário
outevent1443	1	#10	anti-horário

A figura 8 mostra o resultado final para dois cenários: a) Uma plataforma, Spartan-3 FPGA; e b) duas plataformas, Spartan-3 e um microcontrolador PIC18F4620.

No segundo cenário, o nó de rede do modelo do carro3, o qual está situado no microcontrolador, não tem tabela de encaminhamento. O que tem a tabela de encaminhamento é a ponte e ambos têm o mesmo endereço de rede.

O sistema apresentado terá um comportamento como o apresentado no diagrama de sequência da figura 9.

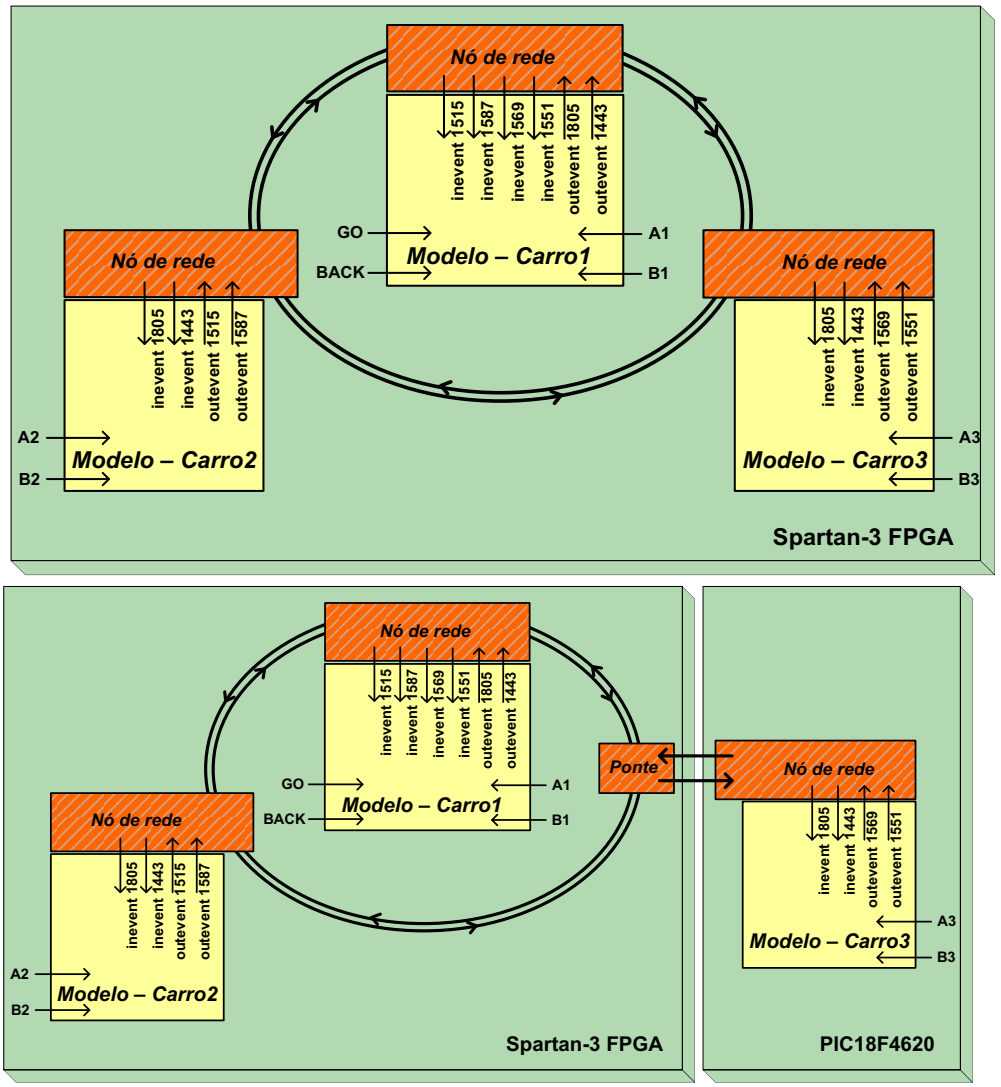


Figura 8. Topologia da rede, a) cenário com base numa única plataforma FPGA e b) cenário com base em duas plataformas heterogéneas (FPGA e PIC)

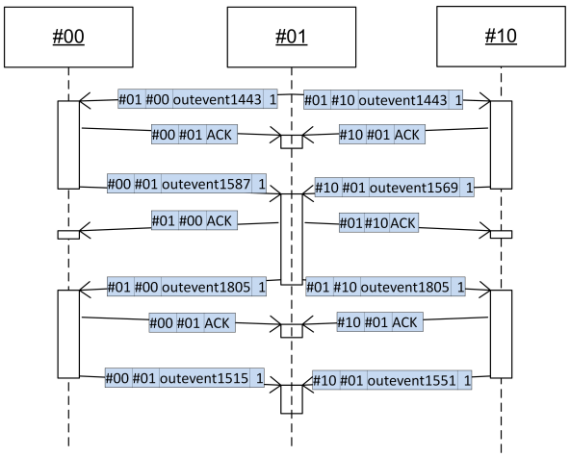


Figura 9. Diagrama de sequência do sistema

4. Conclusões e trabalho futuro

As regras e considerações apresentadas neste artigo permitem a interligação de diversos componentes inicialmente expressos através de modelos de redes de Petri IOPT, através de uma rede de comunicação baseada num duplo anel satisfazendo o protocolo de comunicação série RS-232 (embora operando com ritmos de transmissão mais elevados). Assim, e apesar de ser uma solução de suporte à comunicação potencialmente mais lenta que outras soluções NoC, é uma solução muito flexível que permitirá a sua utilização num número elevado de plataformas em que se pretendam instalar os componentes associados aos submodelos concorrentes, bem como a interligação a soluções proprietárias que satisfaçam o protocolo RS-232.

A metodologia proposta beneficia das vantagens associadas às metodologias de desenvolvimento baseadas em modelos, permitindo um tempo de desenvolvimento potencialmente mais curto e sendo

menos sensível a erros de codificação, uma vez que se utilizam ferramentas de geração automática de código. Contribui, também, para a utilização das ferramentas geradas no projecto FORDESIGN e para o reforço da utilização de redes de Petri no desenvolvimento de sistemas embutidos com base em plataformas reconfiguráveis, mantendo-se a capacidade de interligação a sistemas externos (heterogéneos).

O conjunto de regras de construção da rede foram definidas (nomeadamente o método para atribuição do endereço a cada um dos nós) e a solução validada através dos exemplos apresentados. O passo seguinte é o de desenvolver uma ferramenta que permita automatizar o processo, aplicando as regras descritas e os *templates* criados de forma a gerar automaticamente o código necessário para cada plataforma.

Será também objecto de trabalho futuro a comparação com outros tipos de *Networks-on-Chip* e a identificação de regras que permitam a construção de uma topologia híbrida conforme o modelo inicial sem nunca descurar a ligação com sistemas externos, nomeadamente computadores de utilização geral e microcontroladores de baixo custo.

Referências

[1] J. Nurmi, "Network-on-Chip: A New Paradigm for System-on-Chip Design," System-on-Chip, 2005. Proceedings. 2005 International Symposium on , vol., no., pp.2-6, 17-17 Nov. 2005

[2] D. Bertozzi and L. Benini, "Xpipes: a network-on-chip architecture for gigascale systems-on-chip," IEEE Circuits and Svstems Magazine, vol 4, issue 2, 2004, pp. 18-31.

[3] D. Sigüenza Tortosa, T. Ahonen, and J. Nurmi, "Issues in the Development of a Practical NoC: The Proteo Concept," Integration, The VLSI Journal, Elsevier, Vol. 38, pp. 95-105, October 2004.

[4] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch, "The Nostrum backbone-a communication protocol stack for Networks on Chip," in Proc. International Conference on VLSI Design, 2004, pp. 693-696.

[5] D. Wiklund and D. Liu,"SoCBUS: Switched network on chip for hard real time embedded sytems," in Proc. International Symposium on Parallel and Distributed Processing, April 22-26., 2003.

[6] P. Guerrier and A. Greiner, "A generic architecture for on-chip packet-switched interconnections," In Proc. DATE 2000, Paris, France, March 27-30, 2000, pp.250-256.

[7] H. Kariniemi and J. Nurmi, "Reusable XGFT interconnect IP for Network-on-Chip implementations," in Proc. International Symposium on System-on-Chip, Nov. 16-18, 2004, pp. 95-102.

[8] X. Wang and J. Nunni, "An On-Chip CDMA Communication Network," in Proc. International Symposium on System-on-Chip, Tampere, Finland, November 15-17, 2005.

[9] K. Goossens, J. Dielissen, and A. Radulescu, "Ethereal Network-on-Chip: Concepts, architectures and implementations," IEEE Design & Test of Computers, vol 22, no. 5, 2005, pp. 414421.

[10] L. Gomes and J.-M. Fernandes (Eds), Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation. Publisher: IGI Global; Information Science Reference, 2009.

[11] D. Gajski, S. Abdi, A. Gerstlauer, and G. Sshirner, Embedded System Design Modeling, Synthesis and Verification. Springer, 2009.

[12] FORDESIGN Project home page. http://www.uninova.pt/fordesign, 2009

[13] L. Gomes, J.-P. Barros, A. Costa, R. Nunes: The input-output place-transition Petri net class and associated tools. In Proceedings of INDIN'2007 - 5th IEEE International Conference on Industrial Informatics. IEEE Computer Society Press, July 2007, pp. 23–26.

[14] A. Costa, L. Gomes, "Petri net Partitioning Using net Splitting Operation," in Proceedings of INDIN'2009 (7th IEEE International Conference on Industrial Informatics), 2009.

[15] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri net markup language: Concepts, technology, and tools. In W. van der Aalst and E. Best, editors, Proceeding of the 24th International Conference on Application and Theory of Petri Nets, volume 2679 of LNCS, pages 483{505, Eindhoven, Holland, jun 2003. Springer-Verlag.

[16] M. Silva, Las Redes de Petri: en la Automática y la Informática. Editorial AC, Madrid, 1985.

Uma Linguagem para Geração Automática de Arquiteturas Baseadas em Computação Reconfigurável

Ricardo Menotti*, João M. P. Cardoso†, Marcio M. Fernandes‡, Eduardo Marques⋆
*Coordenação de Informática - Universidade Tecnológica Federal do Paraná
Campo Mourão - Brasil
†Faculdade de Engenharia - Universidade do Porto
Porto - Portugal
‡Departamento de Computação - Universidade Federal de São Carlos
São Carlos - Brasil
⋆Instituto de Ciências Matemáticas e de Computação - Universidade de São Paulo
São Carlos - Brasil
menotti@utfpr.edu.br, jmpc@acm.org, marcio@dc.ufscar.br, emarques@icmc.usp.br

Abstract

Field-Programmable Gate Arrays (FPGAs) are becoming increasingly important in embedded and high-performance computing systems. They have shown important speedups and allow solutions between the efficiency of Application-Specific Integrated Circuits (ASICs) and the flexibility of microprocessors. However, to program efficiently FPGAs, one needs the expertise of hardware developers and to master hardware description languages (HDLs) such as VHDL or Verilog. The attempts to furnish a high-level compilation flow (e.g., from C programs) have generically block in the difficulties to achieve efficient results. Bearing in mind the FPGA resources, we developed LALP, a novel language to program FPGAs. The new language is supported by mapping techniques that are being integrated in a compiler. The main idea behind LALP is to provide a higher abstraction level than HDLs, to exploit the intrinsic parallelism of the hardware resources and to permit the programmer to control execution stages whenever the compiler techniques are unable to generate efficient implementations. In this paper we describe LALP and show how it can be used to achieve high-performance computing solutions.

1. Introdução

A computação reconfigurável tem se mostrado uma interessante alternativa para o desenvolvimento de sistemas em que o alto desempenho e o baixo consumo de energia são requeridos. As características de reconfigurabilidade dos dispositivos deste tipo permitem que os sistemas possuam desempenho próximo aos obtidos com *hardware* dedicado enquanto mantêm a versatilidade das soluções baseadas em software. Isso ocorre principalmente porque os dispositivos deste tipo, entre os quais, pode-se destacar os FPGAs *(Field-Programmable Gate Array)*, possuem uma enorme quantidade de componentes que podem ser utilizados para compor arquiteturas capazes de atingir altos níveis de paralelismo por meio da execução em *pipelining*. Além dos blocos lógicos reconfiguráveis, os FPGAs atuais possuem outros componentes importantes tais como os blocos de DSP *(Digital Signal Processing)* e as memórias de diferentes tamanhos e características, espalhados no dispositivo.

Embora os dispositivos reconfiguráveis permitam a construção de sistemas eficientes, o processo de desenvolvimento utilizado requer o domínio de linguagens de descrição de *hardware*, como por exemplo Verilog e VHDL, além de experiência em desenvolvimento de circuitos integrados. Muitos esforços têm sido realizados na tentativa de obter automaticamente arquiteturas especializadas a partir de descrições em alto nível, como programas descritos em C ou Java [1, 2]. A maioria das abordagens se baseia em técnicas de escalonamento adaptadas das técnicas de *software pipelining* utilizadas com microprocessadores e raramente resultam em sistemas eficientes. Isso ocorre porque as técnicas utilizadas, entre as quais a de *modulo scheduling*[3] merece destaque, são fortemente baseadas nos recursos disponíveis na arquitetura alvo e não consideram a diversidade de recursos presentes nos FPGAs.

Neste trabalho é apresentada uma linguagem de alto nível e seu respectivo compilador para a geração automática de arquiteturas especializadas para execução em dispositivos reconfiguráveis (FPGAs). O objetivo da técnica é oferecer uma alternativa no processo de desenvolvimento quando as ferramentas de síntese de alto nível[4] não são capazes de gerar sistemas que atendam os requisitos de desempenho. Para tal, a linguagem oferece recursos capazes de orientar a geração do *hardware* em termos de escalonamento, mantendo um alto nível de abstração.

2. LALP

A técnica utilizada na geração das arquiteturas especializadas é denominada ALP (*Aggressive Loop Pipelining*) [5, 6] e utiliza uma biblioteca de componentes VHDL,

como muitos outros compiladores de *hardware*. A idéia central é a utilização de contadores para implementar os *loops* no código, sejam eles seqüenciais ou aninhados. Os contadores possuem sinais de controle que habilitam a execução das operações no ciclo correto, como a execução predicativa presente em algumas arquiteturas. Os sinais de controle, bem como os valores de cada contador, são propagados ao longo da arquitetura por meio de registradores de deslocamento. Um conjunto de algoritmos foi desenvolvido para facilitar a obtenção dos parâmetros corretos de escalonamento a serem inseridos na arquitetura.

Para facilitar a criação e a ligação dos componentes utilizados na técnica desenvolvida foi criada a linguagem LALP (*Language for Aggressive Loop Pipelining*) [7]. A linguagem foi concebida com uma sintaxe semelhante a da linguagem C e Java, no que diz respeito as operações lógicas e aritméticas, mas com construtores simplificados para instruções condicionais e de repetição. O compilador da linguagem foi desenvolvido com a ajuda do JavaCC [8], uma ferramenta para a criação de *parsers* e compiladores em Java.

Para demonstrar as funcionalidades da linguagem, será usado um exemplo simples que calcula a soma do produto de dois vetores. O Código 1 apresenta o código fonte deste exemplo para a linguagem C.

Código 1. Algoritmo para calcular a soma do produto de dois vetores descrito em linguagem C

```
1  #define N 2048
2
3  int dotprod() {
4    int x[N], y[N];
5    int i, sum;
6    sum = 0;
7    for (i=0; i<N; i++)
8      sum += x[i] * y[i];
9    return sum;
10 }
```

O Código 2 apresenta a descrição em LALP para geração de uma arquitetura capaz de calcular a mesma soma.

Código 2. Descrição do mesmo algoritmo em LALP

```
1  const DATA_WIDTH = 32;
2  const ITERATIONS = 2048;
3
4  typedef fixed(DATA_WIDTH, 1) int;
5  typedef fixed(1, 0) bit;
6
7  dotprod_alp(out int sum, out bit done,
8              in bit init) {
9    {
10     int x[ITERATIONS], y[ITERATIONS];
11     int acc;
12     fixed(16, 0) i;
13   }
14   counter (i=0; i<ITERATIONS; i++@1);
15   i.clk_en = init;
16   x.address = i;
17   y.address = i;
18   acc += x.data_out * y.data_out
19          when i.step@1;
20   sum = acc;
21   done = i.done@2;
22 }
```

Nas linhas 1 e 2 são declaradas constantes, utilizadas para o número de bits de cada valor e para o número de iterações, respectivamente. Na linha 4 é definido um tipo de dado utilizado para 32 bits de ponto fixo com sinal e na linha seguinte um tipo de um único bit para sinais de controle. A linha 7 inicia com um nome que será utilizado na criação da entidade em VHDL, seguido de sinais de entrada e saída desta entidade. O bloco que vai da linha 9 até a linha 13 contém declarações de variáveis escalares e arranjos.

As instruções propriamente ditas iniciam com o contador na linha 14. A diretiva `@1` indica que o componente irá gerar um novo valor a cada ciclo de relógio e poderia ser omitida, pois este é o valor padrão. Em casos em que ocorrem dependências entre as iterações, um valor diferente pode ser necessário. Na linha 15 o sinal externo de inicialização `init` é utilizado para habilitar a contagem. As linhas seguintes indicam que o endereçamento dos vetores será determinado pela variável `i`. Estas instruções podem ser facilmente substituída por uma macro nas formas `x[i]` e `y[i]`.

O compilador ALP assume que o código está na forma SSA (*Static Single Assignment*), e portanto cada componente recebe valores de uma única origem. Caso haja mais de uma atribuição para a mesma variável, será necessário informar por meio da cláusula `when` o momento das atribuições seguintes para que seja gerado um componente multiplexador. A linha 18 descreve as operações principais do código que devem aguardar um ciclo após o início da contagem. Finalmente, a linha 20 indica que o sinal `sum` irá externar a soma dos valores e o sinal `done` do contador também será apresentado como um pino de saída da entidade. O compilador possui ainda um diretiva, por meio da linha de comandos, capaz de gerar saídas para todas as portas dos componentes, o que pode ser de grande utilidade para fins de depuração do *hardware* gerado.

O *hardware* obtido a partir do Código 2 é apresentado na Figura 1. O escalonamento apresentado sugere que cada iteração necessita de 3 ciclos para ser completada, mas como não há dependências entre elas, uma nova iteração é iniciada a cada ciclo.

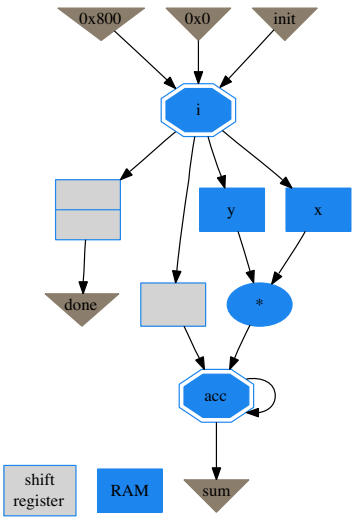


Figura 1. *Hardware* gerado a partir do Código 2

Uma versão alternativa do algoritmo é apresentado no Código 3, registrando-se as saídas das memórias e

978-972-789-304-1 REC'2010

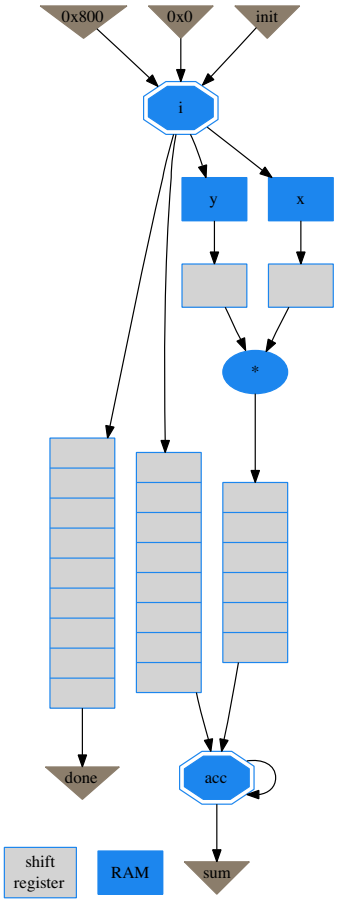


Figura 2. *Hardware* gerado a partir do Código 3

adicionando-se seis estágios de *pipelining* ao multiplicador. Nesta versão cada iteração demora 8 ciclos para ser completada mas a frequência máxima obtida é superior. A Figura 2 apresenta a implementação alternativa, na qual registradores de deslocamento são adicionados para obter a sincronização correta das operações.

Código 3. Versão modificada para melhorar o desempenho em termos de frequência máxima

```
1  const DATA_WIDTH = 32;
2  const ITERATIONS = 2048;
3
4  typedef fixed(DATA_WIDTH, 1) int;
5  typedef fixed(1, 0) bit;
6
7  dotprod_alp(out int sum, out bit done,
8              in bit init) {
9    {
10     int x[ITERATIONS], y[ITERATIONS];
11     int acc;
12     fixed(16, 0) i;
13   }
14   counter (i=0; i<ITERATIONS; i++@1);
15   i.clk_en = init;
16   x.address = i;
17   y.address = i;
18   acc += ((x.data_out@1) * (y.data_out@1)@6)
19          when i.step@8;
20   sum = acc;
21   done = i.done@9;
22 }
```

As visualizações das Figuras 1 e 2 apresentam um formato mais simplificado, ideal para sincronizar as operações durante o processo de desenvolvimento caso seja necessário. Neste formato os pinos de entrada/saída e as constantes são representados por triângulos, os componentes não registrados por elipses e os componentes registrados por octógonos.

Existe ainda a possibilidade de se gerar representações em outro formato, mais voltado para os detalhes de cada componente e suas conexões, apresentado na Figura 3. A partir desta visualização é possível construir o código VHDL completo da arquitetura, pois todos os nomes e tipos utilizados no código são representados. Os pinos de entrada/saída são representados formas retangulares e as constantes por elipses. Os componentes registrados, possuem coloração cinza enquanto os não registrados são brancos. Dentro de cada componente as portas de entrada são apresentadas na parte superior e as de saída na parte inferior. A coloração e a nomenclatura (nomes com colchetes) são utilizadas nas arestas para diferenciar sinais simples (`std_logic`) de sinais compostos (`std_logic_vector`), embora sinais compostos possam ter somente um bit. Ambas as visualizações são geradas com a ferramenta Graphviz [9] a partir de descrições textuais.

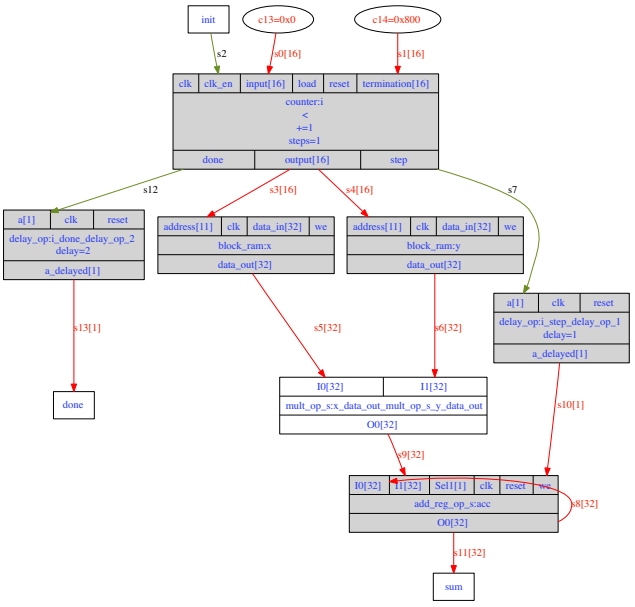


Figura 3. Visualização alternativa apresentando componentes detalhados

LALP difere das linguagens tradicionais de programação no fato de que assume que as instruções serão executadas em paralelo, exceto quando ocorrem dependências. Por padrão, as atribuições são registradas e as sub-expressões são diretas. Além da função de especificar o número de ciclos entre valores consecutivos produzidos pelos contadores, o símbolo @ também tem a função de especificar se uma atribuição/operação será registrada e em quantos ciclos isso irá ocorrer. A Figura 4 apresenta alguns exemplos de utilização desta diretiva. A inicialização das variáveis escalares e dos arranjos é realizada da mesma forma da linguagem C e podem ser descritas normalmente conforme o Código 4. Para os arranjos são gerados arquivos VHDL com valores iniciais. Na implementação

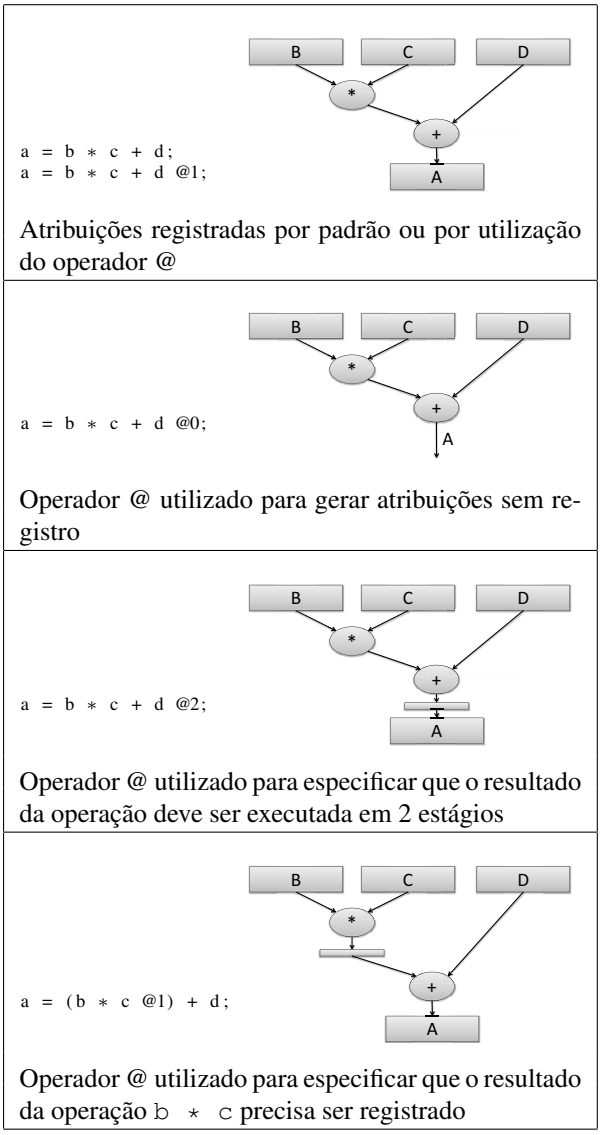


Figura 4. Exemplos do uso do operador @ em expressões

atual do compilador, é considerada uma memória interna distinta no FPGA por arranjo. Caso o programador deseje mapear vários arranjos em uma mesma memória deverá agrupá-los manualmente. O mapeamento de múltiplos arranjos em memória pode ser adicionado até mesmo por um pré-processamento do código antes da compilação em versões futuras.

Código 4. Inicialização de variáveis

```
1  int indexTable[16] = {
2    -1, -1, -1, -1, 2, 4, 6, 8,
3    -1, -1, -1, -1, 2, 4, 6, 8
4  };
5  bit bufferstep = 1;
```

Em LALP é possível descrever contadores aninhados e em série, sem a necessidade de se utilizar blocos como na linguagem C. O Código 5 apresenta as repetições do algoritmo da FDCT (*Fast Discrete Cosine Transform*) descritas em linguagem C e no Código 6 as mesmas repetições são descritas em LALP. A linha 16 indica que o contador k só será iniciado 17 ciclos após o contador i terminar, obtendo-se assim o mesmo efeito de seqüência.

Código 5. Exemplo de repetições na linguagem C

```
1   i_1 = 0;
2   for (i = 0; i < num_fdcts; i++) {
3     for (j = 0; j < N; j++) {
4       f0 = dct_io_ptr[ 0+i_1];
5       f1 = dct_io_ptr[ 8+i_1];
6       ...
7       i_1++;
8     }
9     i_1 += 56;
10  }
11  i_1 = 0;
12  for (k = 0; k < N*num_fdcts; k++) {
13    ...
14  }
```

Código 6. Exemplo de repetições na linguagem LALP

```
1  counter (i=0; i<num_fdcts; i+=64@72);
2  i.clk_en = init;
3  i_plus_8 = i + 8;
4    counter (j=i; j<i_plus_8; j++@9);
5    j.clk_en = init;
6    j.load = i.step;
7    j_plus_64 = j + 64;
8      counter (i_1=j; i_1<j_plus_64; i_1+=8);
9      i_1.clk_en = init@2;
10     i_1.load = j.step;
11     dct_io_ptr.address = i_1;
12     f0 = dct_io_ptr.data_out when (j.step@3);
13     f1 = dct_io_ptr.data_out when (j.step@4);
14     ...
15 counter (k=0; k<num_fdcts; k++);
16 k.clk_en = i.done@17;
17   ...
```

3. Resultados

Nesta seção apresentam-se resultados obtidos considerando um FPGA Xilinx Virtex5 (XC5VLX30-3FF324) e utilizando as ferramentas da Xilinx (ISE 9.2i) para implementar no FPGA as arquiteturas descritas em VHDL e obtidas a partir de código LALP.

Uma das grandes vantagens do uso de LALP é a possibilidade de exploração dos níveis de *pipelining* mais adequados para cada arquitetura, dependendo do desempenho desejado e dos recursos disponíveis no dispositivo alvo. Os compiladores utilizados para síntese de alto nível oferecem pouca ou nenhuma possibilidade de interferir no processo de compilação. Por outro lado, ajustes deste tipo podem ser complexos se realizados diretamente nas linguagens de descrição de *hardware*, pois a mudança no número de ciclos de um componente pode interferir na sincronização das operações. A Figura 5 apresenta configurações do exemplo *Dotprod* com diferentes estágios de *pipelining* para o multiplicador. As duas últimas configurações adicionam ainda registradores antes do multiplicador e atingem um ganho ainda maior em termos de frequência e, consequentemente, em tempo de execução.

A exploração do número de níveis de *pipeline* do multiplicador utilizado permitiu acelerar em 3,8x o tempo de execução do *Dotprod* precisando para isso de 1,6x mais LUTs e 1,5x mais registradores. É de notar que esta exploração é conseguida alterando valores @ e gerando VHDL por cada alteração, fato que pode facilitar a exploração automática através de *scripts*.

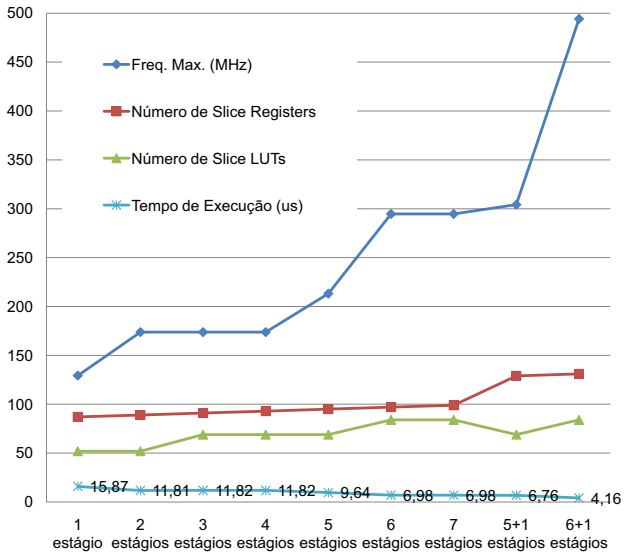


Figura 5. *Dotprod* com diferentes configurações

Código 7. Implementação do filtro Sobel em C

```
#define cols 10
#define rows 10
#define N cols*rows

int main() {
  char in[N];
  char out[N];
  int H, O, V, i;
  int i00, i01, i02;
  int i10,      i12;
  int i20, i21, i22;

  for (i = 0; i < cols*(rows-2)-2; i++) {
    i00=in[i];
    i01=in[i+1];
    i02=in[i+2];
    i10=in[i+cols];
    i12=in[i+cols+2];
    i20=in[i+2*cols];
    i21=in[i+2*cols+1];
    i22=in[i+2*cols+2];
    H = -   i00 - 2*i01 -   i02 +
        +   i20 + 2*i21 +   i22;
    V = -   i00         +   i02
        - 2*i10         + 2*i12
        -   i20         +   i22;
    if (H<0)
      H = -H;
    if (V<0)
      V = -V;
    O = H + V;
    if (O > 255) O = 255;
      out[i + 1] = (char)O;
  }
  return O;
}
```

Os Códigos 7 e 8 apresentam um comparativo entre as linguagens C e LALP na implementação do filtro *Sobel*. A versão VHDL foi omitida por questões de espaço.

A Tabela 1 ilustra o número de linhas de código C, código LALP e código VHDL gerado a partir do código LALP para os exemplos *Dotprod* e *Sobel*. Pode-se ver que em termos de número de linhas, a linguagem LALP não é tão compacta como a linguagem C. No entanto o código VHDL gerado a partir de LALP tem complexidade em termos de número de linhas muito maior (cerca de 10x para o *Dotprod* e 30x para o *Sobel*). Esse código VHDL descreve a estrutura da arquitetura em termos de componentes existentes na biblioteca de componentes. Essa biblioteca de componentes utiliza descrições RTL a nível comportamental.

Código 8. Implementação do filtro Sobel em LALP

```
const DATA_WIDTH = 16;
const COLS = 10;
const N 100;

typedef fixed(DATA_WIDTH, 1) int;
typedef fixed(1, 0) bit;
typedef fixed(8, 0) byte;

sobel_alp(in bit init, out bit done) {
  {
    int H, O, V, Hpos, Vpos, Otrunk;
    int i, addr;
    int i00, i01, i02;
    int i10,      i12;
    int i20, i21, i22;
    int in[N];
    int out[N];
  }
  counter (i=0; i<78; i+=1@8);
  i.clk_en = init;
  in.address = addr;
  addr = i;
  addr = (i@1) + 1 when i.step@1;
  addr = (i@2) + 2 when i.step@2;
  addr = (i@3) + COLS when i.step@3;
  addr = ((i@4) + COLS) + 2 when i.step@4;
  addr = ((i@5) + COLS) + COLS when i.step@5;
  addr = (((i@6) + COLS) + COLS) + 1 when i.step@6;
  addr = (((i@7) + COLS) + COLS) + 2 when i.step@7;
  i00 = in when i.step@2;
  i01 = in when i.step@3;
  i02 = in when i.step@4;
  i10 = in when i.step@5;
  i12 = in when i.step@6;
  i20 = in when i.step@7;
  i21 = in when i.step@8;
  i22 = in when i.step@9;
  H = ((-i00) + (-2*i01)) +
    (((-i02) + i20) +
    (2*i21 + i22));
  V = ((-i00) + i02) +
    (((-2*i10) + 2*i12) +
    ((-i20) + i22));
  Hpos = H < 0 ? -H : H;
  Vpos = V < 0 ? -V : V;
  O = Hpos + Vpos;
  Otrunk = 255;
  Otrunk = O when O > 255;
  out.data_in = Otrunk when i.step@13;
  out.address = i@13;
  done = i.done@13;
}
```

Tabela 1. Linhas de código

Exemplo	#linhas		
Linguagem	C	LALP	VHDL
Dotprod	10	22	225
Sobel	36	52	1298

LALP tem sido utilizada para mapear vários exemplos em FPGAs. Os resultados alcançados e previamente publicados em [6, 7] permitem concluir que LALP oferece uma alternativa credível sempre que o programador não consiga alcançar o desempenho pretendido com o uso de fluxos de compilação que traduzem subconjuntos de código C em Verilog ou VHDL. Acredita-se que nesses casos é mais fácil desenvolver o acelerador em LALP do que em Verilog ou VHDL, embora evidências fortes para esta afirmação necessitem de ser analisadas com o estudos com vários programadores.

O mapeamento das *benchmarks Autcor*, *ADPCM Decoder*, *ADPCM Coder*, *Bubble Sort*, *Dotprod*, *Fibonacci*, *Max*, *Sobel* e *Vecsum* [10, 11, 12] usando LALP permitiu atingir melhorias de desempenho na ordem de 2,8x (valor médio) em relação à ferramenta comercial C to Verilog [13]. Para as *benchmarks* referidas foram obtidas melhorias mínimas de 1,4x e máximas de 7,1x. Ainda relevante é o fato de o número de recursos *hardware* necessários ser inferior usando LALP. Tal deve-se ao fato da técnica de *loop pipelining* utilizada ser muito mais propicia quando existem recursos que permitem execução horizontal e vertical e devido ao fato de LALP utilizar eficientemente recursos do FPGA, como os registos de deslocamento, por exemplo. A utilização de LALP para mapear o *ADPCM Decoder* e o *Encoder* em um FPGA Stratix III (utilizando como *back-end* a ferramenta Quartus II 8.1 da Altera) permitiu acelerar estes dois exemplos em 75,8x e 20,3x em relação à execução no processador PowerPC embebido no FPGA Virtex-2Pro a executar a 100 MHz, respectivamente.

4. Conclusões

Este artigo apresentou a LALP, uma linguagem de domínio específico para programar aceleradores de *hardware* para mapeamento em sistemas baseados em FPGA. Foram apresentadas as principais características da linguagem, bem como sua semântica, incluindo exemplos ilustrativos de utilização. A linguagem foi desenvolvida para explorar a capacidade de se obter múltiplos fluxos de controle e execução paralela, inerentes dos sistemas reconfiguráveis. Os resultados obtidos com LALP são muito encorajadores, especialmente pelo fato de terem sido obtidos melhores desempenhos usando um menor número de recursos de hardware em comparação com os resultados obtidos por meio de compiladores para *hardware* convencionais. Os programas codificados em LALP podem ser automaticamente traduzidos pelo compilador para descrições VHDL prontas para síntese RTL.

Atualmente, o trabalho se concentra em desenvolver algoritmos para sincronização automática ou sugestão de parâmetros ao programador quando possíveis conflitos ocorrerem. O objetivo é que todas ou quase todas as diretivas @ sejam inferidas e que só se façam necessárias quando houver a necessidade de explorar graus de paralelismo diferentes, seja por requisitos de área ou frequência máxima.

5. Agradecimentos

Os autores Ricardo Menotti, Marcio M. Fernandes e Eduardo Marques agradecem ao CNPq e à FAPESP pelo financiamento ao Instituto Nacional de Ciência e Tecnologia em Sistemas Embarcados Críticos (INCT-SEC), processos 573963/2008-8 e 08/57870-9. O autor João M. P. Cardoso agradece o apoio financeiro concedido pela FCT através do projeto COBAYA (PTDC/EEA-ELC/70272/2006).

Referências

[1] João Manuel Paiva Cardoso and Pedro C. Diniz. *Compilation Techniques for Reconfigurable Architectures*. Springer Publishing Company, Incorporated, 2008.

[2] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer Publishing Company, 1st edition, 2008.

[3] B. Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, New York, NY, USA, 1994. ACM Press.

[4] D. D. Gajski, N. D. Dutt, A. C. H. Wu, and S. Y. L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[5] Ricardo Menotti, Eduardo Marques, and João Manuel Paiva Cardoso. Aggressive Loop Pipelining for Reconfigurable Architectures. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 501–502, 2007.

[6] Ricardo Menotti, João Manuel Paiva Cardoso, Márcio Merino Fernandes, and Eduardo Marques. Automatic Generation of FPGA Hardware Accelerators Using a Domain Specific Language. In *FPL 2009-International Conference on Field Programmable Logic and Applications*, pages 457–461, 2009.

[7] Ricardo Menotti, João Manuel Paiva Cardoso, Márcio Merino Fernandes, and Eduardo Marques. LALP: A Novel Language to Program Custom FPGA-based Architectures. In *Proceedings of the 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 3–10, Los Alamitos, CA, USA, 2009. IEEE Computer Society Press.

[8] Viswanathan Kodaganallur. Incorporating language processing into Java applications: A JavaCC tutorial. *IEEE Softw.*, 21(4):70–77, 2004.

[9] AT&T Research. *Graphviz: Graph Visualization Software*, 2006.

[10] T. R. Halfhill. EEMBC releases first benchmarks. *Microprocessor Report*, 1, 2000.

[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[12] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335. IEEE Computer Society, 1997.

[13] C-to-Verilog.com. *C-to-Verilog*, 2009.

Sessão Regular 2

Telecomunicações I

Moderação: Nuno Roma
Instituto Superior Técnico / INESC-ID

Implementação de Algoritmos em FPGA para Estimação de Sinal em Sistemas Ópticos Coerentes

Nuno M. Pinto
DEEC, FEUP
ee03171@fe.up.pt

Henrique M. Salgado
INESC Porto, FEUP
hsalgado@inescporto.pt

João C. Ferreira
INESC Porto, FEUP
jcf@fe.up.pt

Luís M. Pessoa
INESC Porto, FEUP
luis.pessoa@ieee.org

Resumo

Neste artigo descreve-se a implementação em FPGA de algoritmos para estimação de sinal em sistemas ópticos coerentes. A ferramenta de desenvolvimento usada para a implementação destes em hardware foi o System Generator. Foram criadas duas implementações dos algoritmos estudados, uma sequencial e outra paralela. Obtiveram-se resultados em termos de taxas de transmissão e do desempenho de cada uma das implementações, que permitiram avaliar a utilização de recursos na FPGA.

Foram, ainda, implementados compensadores para 200 km e 500 km de dispersão para a fibra óptica SSMF. Cada equalizador, juntamente com o compensador, foi testado para sistemas de transmissão com modulação 4QAM e 16QAM. Os resultados são bastante promissores justificando a aposta nesta tecnologia.

1. Introdução

Os sistemas ópticos coerentes têm vindo a ganhar importância nas comunicações por fibra óptica. Uma grande vantagem destes sistemas, comparativamente aos sistemas de Modulação em Intensidade e Detecção Directa (IM/DD), consiste na possibilidade de usar na transmissão vários tipos de modulação, nomeadamente a modelação em fase (M-PSK) e constelações multi-nível (M-QAM), havendo a preservação da fase do campo eléctrico do domínio óptico para o domínio eléctrico, sendo o sinal amostrado à taxa de *Nyquist*. Adicionalmente, é possível compensar as penalidades lineares do canal de transmissão, bem como a dispersão cromática (CD) e a dispersão do modo de polarização (PMD) através de um filtro linear [6], que pode operar de forma adaptativa para superar as distorções do sinal ao longo do tempo. Estes sistemas ganharam um renovado interesse devido à disponibilidade de Processamento Digital de Sinal (PDS) de alta velocidade, o que permite que operações complexas sejam realizadas no domínio digital, dando origem a um receptor óptico reconfigurável.

Os conversores analógico-digitais serão capazes de satisfazer brevemente as elevadas taxas de amostragem requeridas em sistemas de transmissão ópticos de alta velocidade. A evolução é no sentido de se usar conversores de elevado desempenho (>40 GSample/s) que aliados a FPGAs *(Field Programmable Gate arrays)* tornam estes sistemas realizáveis em tempo real. Existem já no mercado conversores de 30 GSample/s, com capacidades especiais para interligação com FPGA [4] [11].

A implementação em FPGA é uma boa opção para estes sistemas, uma vez que é uma plataforma de aplicação muito flexível e moldável à situação pretendida. De facto, foram realizadas várias experiências de transmissão a alta velocidade recorrendo a FPGAs e conversores rápidos [2] [8] [17] [7].

No receptor, a fase do Oscilador Local (OL) deve ser sincronizada com a fase do sinal, para evitar as dificuldades associadas ao uso de uma OPLL *(Optical Phase-Locked Loop)*. Essa sincronização pode ser feita em PDS através de algoritmos de estimação de fase digital, onde o OL fica em funcionamento livre. Os algoritmos apropriados para a estimação de fase e compensação de dispersão foram estudado em [13] [14]. A implementação destes algoritmos usando FPGAs com capacidade de processamento paralelo [9] são discutidas neste artigo. Para este trabalho foram implementados algoritmos com compensação de dispersão adaptativa usando a ferramenta *System Generator* da Xilinx.

Após a introdução, segue-se na secção 2 uma descrição da equalização adaptativa, que retrata os algoritmos usados no trabalho, bem como o funcionamento do módulo para a compensação da dispersão. A secção 3 apresenta a implementação em FPGA e as ferramentas de desenvolvimento. Os resultados são apresentados na secção 4 e, por último, na secção 5 são dadas as conclusões.

2. Equalização adaptativa

Um equalizador adaptativo é um filtro localizado no receptor que recebe os dados provenientes de um canal de transmissão, por exemplo a fibra óptica. A fibra óptica, como qualquer canal, introduz distorções no sinal transmitido. Ora, um equalizador adaptativo vai, através de algoritmos, adaptar-se às características adversas do canal para, deste modo, as poder compensar no sinal transmitido, ou seja, o equalizador vai retirar as distorções que o sinal sofre ao longo do canal dispersivo, ficando este o mais semelhante possível ao sinal originalmente transmitido.

Os algoritmos podem ser auto-adaptativos, como é o caso do algoritmo CMA *(Constant Modulus Algorithm)* cuja característica é convergir para um módulo constante, ou serem supervisionados e necessitando de uma sequência de treino, como é o caso do algoritmo LMS *(Least Mean*

Square) cuja característica é possuir um decisor.

2.1. O Algoritmo CMA

O algoritmo CMA realiza uma equalização cega e foi estudado por Sato em 1975 [15] e mais tarde por Godard em 1980 [5]. O modo de funcionamento deste algoritmo é atingir a convergência perante aplicações com envolvente constante, ou módulo constante [1]. O CMA atinge a convergência adaptando automaticamente os seus coeficientes às características do canal.

Este algoritmo é o mais usado em equalizadores adaptativos, essencialmente por causa da sua robustez e baixa complexidade, efectuando a estimação dos dados através da minimização da função custo pelo método do gradiente descendente [10].

O processo do equalizador CMA passa por três etapas, onde a primeira é dada pela equação seguinte:

$$y(n) = \mathbf{w}^H(n) \cdot \mathbf{u}(n) \tag{1}$$

onde $y(n)$ representa o sinal à saída do equalizador, obtido através da convolução dos coeficientes do equalizador ($\mathbf{w}^H(n)$) com o sinal que se pretende equalizar ($\mathbf{u}(n)$). Em seguida o erro é calculado da seguinte forma:

$$e(n) = y(n) \cdot (R_2 - |y(n)|^2) \tag{2}$$

onde R_2 é uma constante que depende da constelação seleccionada. Para QPSK, o valor de R_2 é unitário.

Por último, a actualização dos coeficientes é dada por:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu \cdot \mathbf{u}(n) e^*(n) \tag{3}$$

onde μ é o passo de adaptação do algoritmo.

Uma característica importante deste algoritmo é a inicialização dos coeficientes, já que dela depende o sucesso da convergência do CMA. A inicialização mais usada (e também utilizada neste trabalho) é a denominada de *center spike*, e consiste em colocar o coeficiente central igual à unidade, enquanto todos os outros coeficientes são colocados a zero.

Apesar de não tomar em consideração a fase do sinal, o que origina a rotação da constelação, este algoritmo apresenta um bom desempenho, permitindo convergir mesmo se houver ruído de fase elevado. Porém, uma vez atingida a convergência há benefícios em comutar para uma arquitectura baseada em decisões dos símbolos estimados — o LMS *(Least Mean Squares)* — onde a fase do sinal é considerada.

2.2. Algoritmo LMS

O algoritmo LMS, o mais popular de todas as estimativas, foi sugerido por Widrow e Hoff em 1959 [18] e consiste simplesmente em substituir os valores médios das variáveis pelos seus valores instantâneos [1].

É um método estocástico de gradiente descendente em que os coeficientes do filtro adaptativo são obtidos por forma a minimizar o erro quadrático médio da diferença entre o valor decidido e o valor estimado do sinal.

Este algoritmo é em tudo semelhante ao CMA excepto no cálculo do erro. Para tal é necessário um módulo extra que realiza a decisão do símbolo à saída do equalizador. O erro pode ser calculado da seguinte forma:

$$e(n) = d(n) - y(n) \tag{4}$$

onde $d(n)$ é o símbolo dado pelo decisor.

Uma outra característica, que representa uma dificuldade deste algoritmo, é a necessidade de inicialização dos coeficientes. Uma forma de efectuar a inicialização é aplicar uma sequência de treino conhecida até atingir a convergência, sendo depois comutado para o modo Dedicado à Decisão, onde fica a funcionar sem qualquer apoio de treino. Contudo há uma maneira mais eficaz de obter a inicialização dos coeficientes do LMS, evitando sequências de treino: usar o CMA numa fase inicial até se atingir a convergência e depois utilizar esses coeficientes para inicializar o algoritmo LMS. Esta abordagem é a usada neste trabalho.

2.3. Compensação da dispersão

A compensação da dispersão pode ser realizada no domínio óptico ou no domínio eléctrico. No domínio óptico é possível realizar a compensação através de fibras com dispersão cromática contrária à do sistema de transmissão. No domínio eléctrico é possível compensar a dispersão cromática com a ajuda de filtros de resposta impulsional finita. Este método diminui a complexidade associada à compensação no nível óptico.

Recorrendo ao processamento de sinal este módulo pode ser implementado em conjunto com os equalizadores adaptativos discutidos anteriormente. Assim, colocando um módulo deste tipo antes dos algoritmos de equalização, LMS ou CMA, é possível compensar a maior parte da dispersão, sendo o equalizador adaptativo responsável pela compensação da dispersão residual e efeitos variantes no tempo como a PMD.

A implementação do módulo de compensação é realizada em FPGA através de um filtro de coeficientes fixos, cujos valores são calculados de acordo com a transformada inversa de *Fourier* da função de transferência da fibra representada pela equação:

$$G(z,\omega) = exp\left(-j\frac{D\lambda^2 z}{4\pi c}\omega^2\right) \tag{5}$$

onde D representa o coeficiente de dispersão da fibra, λ o comprimento de onda, z a distância de transmissão, c a velocidade da luz e ω a frequência angular.

O filtro de compensação da dispersão é dado por um filtro-passa tudo com a característica $1/G(z,\omega)$ e pode ser construído usando tanto filtros digitais recursivos como não-recursivos [16].

3. Implementação em FPGA

3.1. Metodologia de desenvolvimento

Relativamente à programação da FPGA, existe um conjunto de ferramentas de software associado a um fluxo de projecto que proporciona um alto nível de abstracção ao programador, permitindo que este se foque no algoritmo que deseja implementar em vez de se preocupar com os circuitos que serão implementados. Desta forma, a programação do dispositivo pode ser feita ou através de uma linguagem de descrição de hardware (VHDL ou Verilog) ou recorrendo à ferramenta de modelização de sistemas − *System Generator*.

O *System Generator* é uma ferramenta de projecto integrado com FPGAs, que utiliza como suporte de desenvolvimento o *Simulink*, a ferramenta de modelização, simulação e análise de sistemas dinâmicos do MATLAB.

Além do *Simulink*, o *System Generator* utiliza um conjunto de ferramentas para especificar os detalhes de implementação de hardware em dispositivos da Xilinx. Mas é no *Simulink* que o *System Generator* é apresentado sob a forma de uma biblioteca adicional (*Xilinx Blockset*).

A ferramenta de desenvolvimento *System Generator* possibilita ao utilizador desenvolver algoritmos sofisticados e sistemas de processamento de sinal, abstraindo-se de funções complexas de matemática, lógica, memória ou PDS. A biblioteca da Xilinx no *Simulink* possui também blocos que proporcionam interfaces com outras ferramentas, bem como outros que geram automaticamente o código VHDL ou Verilog [20].

No *Simulink* podem ser usados os blocos das bibliotecas *Simulink* em conjunto com os do *System Generator*. No entanto, há que ter em linha de conta alguns aspectos importantes. O subsistema a implementar em hardware deve ser constituído apenas por elementos do *Xilinx Blockset*. As entradas e saídas deste subsistema são obrigatoriamente constituídas por blocos *Gateway In* e *Gateway Out*, respectivamente. Estes blocos definem a fronteira da FPGA no ambiente de simulação e realizam, portanto, a conversão dos dados entre os formatos internos de MATLAB (números de vírgula flutuante) e os formatos usados no processamento PDS em FPGA (vírgula fixa).

A definição da conversão dos dados de entrada deve encontrar um compromisso entre a precisão requerida pelo algoritmo a ser implementado e a utilização de recursos de hardware. Embora o sistema admita operandos com 4096 bits [19], tais dimensões são claramente superiores ao que é utilizável na prática. A utilização de representação em vírgula fixa leva a que seja necessário descartar alguma informação, levando a desvios, seja por *overflow* (nos bits mais significativos) seja por quantização (nos bits menos significativos).

Neste trabalho o número de bits usados para a representação dos dados é 18 bits, sendo que 1 bit é reservado para o sinal, 5 bits para a parte inteira e os restantes 12 bits para a parte fraccionária. O que pesou nesta escolha foi o facto de haver FPGAs que disponibilizam multiplicadores 18×18 dedicados.

A maioria dos blocos do *Xilinx Blockset* permite ao utilizador escolher a precisão que melhor se ajusta ao projecto, mas também é possível deixar o sistema deduzir o número de bits necessário para representar os resultados de cada bloco a partir das características dos respectivos dados de entrada. A propagação automática das dimensões dos dados leva geralmente ao seu crescimento ao longo da cadeia de cálculo (por exemplo, os produtos requerem uma representação com o dobro dos bits dos operandos). Quando existe realimentação de dados (como é o caso para os algoritmos em consideração, como na figura 1), o mecanismo de dedução de dimensões falha, pelo que é imprescindível especificar pelo menos algumas dessas dimensões.

O *System Generator* permite realizar uma co-simulação software/hardware, com a parte do sistema especificada com elementos do *Xilinx Blockset* a ser executada em FPGA, enquanto os outros elementos são simulados em MATLAB. Trata-se de uma forma cómoda de validar parcialmente o hardware desenvolvido sem prescindir das ferramentas e da comodidade associadas ao ambiente MATLAB.

Além do *Simulink*, o *System Generator* utiliza um conjunto de ferramentas para especificar os detalhes de implementação de hardware em dispositivos da família *Xilinx*. Para tal, o *System Generator* utiliza a biblioteca *Xilinx DSP Blockset*, também instalada no *Simulink* e para gerar a *Netlist* optimizada dos módulos PDS invoca automaticamente o *Xilinx Core Generator*. Opcionalmente, pode-se gerar um *testbench* para usar no *ModelSim* ou no *Xilinx ISE Simulator* para aprofundar o nível de detalhe do projecto a implementar.

3.2. Estrutura da Implementação em Hardware

Os algoritmos em estudo foram desenvolvidos em *System Generator*. Foram implementadas duas configurações diferentes para cada algoritmo, uma sequencial e outra paralela. Conforme a própria designação indica, na versão sequencial os dados são tratados em cadeia sequencial, enquanto que a versão paralela realiza o tratamento dos dados em paralelo.

Apresenta-se aqui o diagrama de blocos de cada um dos algoritmos, começando pelo diagrama do CMA ilustrado na figura 1.

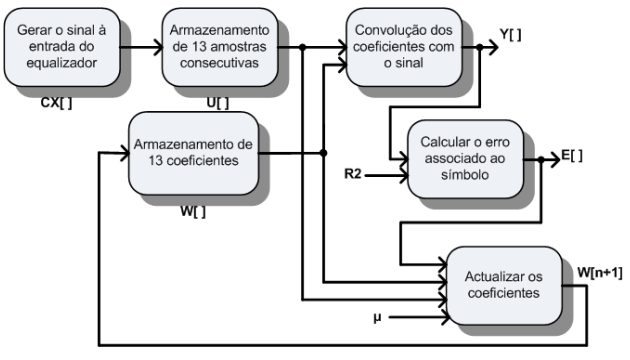


Figura 1. Diagrama do algoritmo CMA.

O CMA armazena quer os dados do sinal, quer os coeficientes do algoritmo. O armazenamento dos dados difere de acordo com a versão a implementar. Na versão sequencial foi utilizado o *Addressable Shift Register (ASR)*, pois o seu simples funcionamento permite apresentar a cada ciclo de relógio uma amostra do sinal, e ainda fazer um deslocamento das amostras descartando as mais antigas. Para a versão paralela foram usados registos individuais interligados, a fim de criar um ASR mas em paralelo. A vantagem deste segundo caso é que as amostras estão disponíveis todas em simultâneo.

Para os coeficientes também foram adoptadas duas abordagens diferentes para cada versão. Na versão sequencial foi usada uma *Dual Port RAM*, cuja vantagem é poder ler e escrever no mesmo ciclo de relógio. Esta vantagem é importante, pois é necessário ler os coeficientes actuais e escrever os novos coeficientes actualizados pelo algoritmo. Já para a configuração paralela o processo usado foi simplesmente um registo controlado por um relógio para garantir que a escrita dos novos coeficientes só é realizável quando os coeficientes antigos já não são necessários.

O número de coeficientes usados pelo algoritmo, bem como o número consecutivo de amostras é de 13, conforme indicado em [16]. As amostras e os coeficientes são utilizados pelo bloco que realiza a convolução dos coeficientes com o sinal, de acordo com a equação 1.

O bloco que calcula o erro associado ao símbolo estimado (pelo bloco do filtro) executa uma operação correspondente à equação 2. Seguidamente são actualizados os coeficientes através da equação (3) e guardados para utilização no próximo ciclo do algoritmo. Também nestes três últimos casos foi necessário criar uma implementação diferente para cada uma das versões implementadas.

A diferença do algoritmo LMS para o anterior é o modo como calcula o erro, que foi implementado como indica o diagrama da figura 2 .

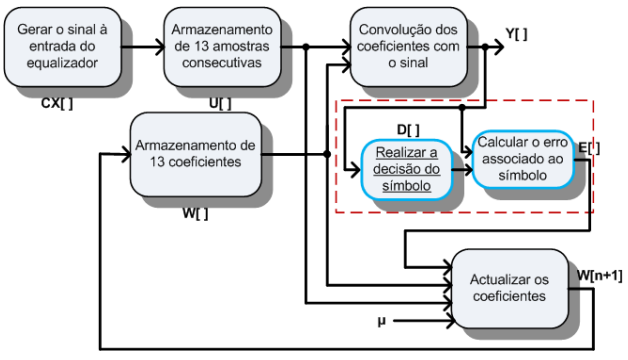


Figura 2. Diagrama do algoritmo LMS.

Importa referir que as amostras do sinal são valores complexos, pelo que é necessário tratar a parte real e imaginária separadamente, uma vez que a FPGA não realiza operações com números complexos. Para tal, quer as amostras, quer os coeficientes são separados em parte real e imaginária, e todas as operações são realizadas tendo em conta essa separação.

4. Resultados obtidos

4.1. Resultados de implementação

Para cada algoritmo (CMA e LMS) foram implementadas duas versões, como já referido. A realização destas duas versões permitiu obter indicações sobre os requisitos dos algoritmos em termos de ocupação da FPGA e sobre as taxas de transmissão que o sistema de transmissão em estudo poderá suportar.

A versão paralela permite naturalmente obter taxas de transmissão mais altas, à custa de uma maior ocupação de recursos na FPGA como se verifica na tabela seguinte.

Por outro lado, apesar de as taxas de transmissão serem mais baixas a implementação da configuração sequencial permite executar o algoritmo em FPGAs com menor número de recursos.

Tabela 1. Comparação entre as versões sequencial e paralela (Virtex-5 XC5VLX330T).

Componentes	**Sequencial**	**Paralela**
Slices	1070	6944
Flip-flops	939	5285
BRAMs	2	0
LUTs	1758	9933
IOBs	108	108
Multiplicadores	14	60
Características		
Max. Frequência	111,669 MHz	108,530 MHz
Min. Período	8,955 ns	9,212 ns
Tempos mínimos		
Latência (períodos)	26	10
Tempo de Símbolo (ns)	232,83	92,12
Taxa de Transmissão (MSymbols/s)	4,29	10,86

A tabela mostra a alocação de recursos, tempos e taxas de transmissão para uma FPGA da família Virtex 5 (XC5VLX330T). É notório que a configuração em paralelo ocupa cerca de 6 vezes mais recursos que a configuração sequencial. No que se refere a multiplicadores dedicados a sua utilização é muito superior.

Um aspecto muito importante e que caracteriza a implementação de cada configuração é a latência, onde se observa que a versão paralela reduz a latência de 26 para 10 períodos em relação à sequencial. Dessa latência depende a taxa de transmissão onde a configuração paralela atinge perto de 11 MSímbolos/s enquanto a sequencial se fica pelos 4 MSímbolos/s.

Esta implementação, tendo sido realizada usando o System Generator, é válida para outros modelos de FPGA, bastando para o efeito gerar uma nova *Netlist*. Esta é uma das grandes vantagens do System Generator e foi o motivo porque se usou esta técnica. Com isso foi possível obter os limites do sistema apresentados na Tabela 2 para várias

plataformas FPGAs em termos de taxas de transmissão.

Tabela 2. Taxas de transmissão em MSímbolos/s para várias plataformas de FPGAs.

Plataforma FPGA	**Taxas de transmissão** Versão Sequencial	Versão Paralela
Spartan 3A XC3S700A	1,92	—
Virtex 4 XC4VLX60	2,23	9,71
Virtex 4 XC4VSX55	2,61	7,49
Virtex 5 XC5VLX330T	4,29	10,86
Virtex 5 XC5VSX95T	4,20	10,11

Em resumo, o sistema de transmissão em estudo pode atingir 10,86 MSímbolos/s na configuração paralela usando a FPGA Virtex 5 XC5VLX330T.

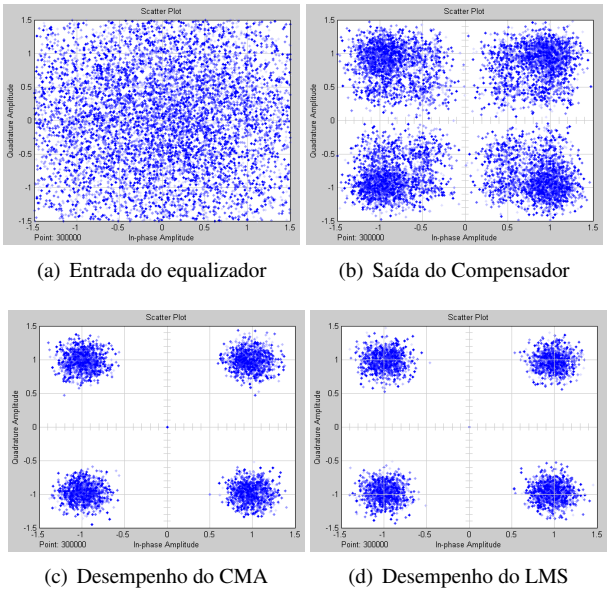
4.2. Resultados de simulação

Para os resultados apresentados em seguida, foram simulados o transmissor e o canal do sistema de transmissão. Para tal foi usada a codificação de impulsos NRZ, obtida através de um *trem* rectangular de impulsos ideais juntamente com um filtro de *Bessel* passa-baixo de 5ª ordem com 3 dB de largura de banda, a 80% da taxa de transmissão dos símbolos. É usado também um filtro *anti-alias*, constituído por um filtro de *Bessel* passa baixo de 3ª ordem.

Os resultados foram obtidos para a modulação 4QAM e posteriormente 16QAM, com 200 km de fibra óptica. Para cada tipo de constelação obtiveram-se as constelações à entrada do equalizador e à saída do compensador. Obtiveram-se ainda as seguintes constelações que demonstram o desempenho quer do algoritmo CMA, quer do algoritmo LMS.

Na figura 3(a) estão representados os símbolos após viajarem através de fibra óptica com 200 km de comprimento para a constelação 4QAM, onde se observa que os dados aparecem bastante distorcidos. Para retirar a dispersão cromática referente a esses 200 km de fibra é necessário um compensador de dispersão constituído por coeficientes fixos cujo número de coeficientes varia consoante a distância de fibra óptica. Para o comprimento de 200 km são necessários 13 coeficientes para obter compensação, como é visível na figura 3(b).

Os algoritmos adaptativos descritos nas secções anteriores são capazes, por si só, de compensar a dispersão cromática da fibra até 200 km de comprimento, quando 13 coeficientes são utilizados; no entanto a rapidez de convergência dos algoritmos é afectada. Daí a necessidade de desenvolver este filtro compensador de coeficientes fixos para 200 km de fibra, a fim de melhorar os tempos de



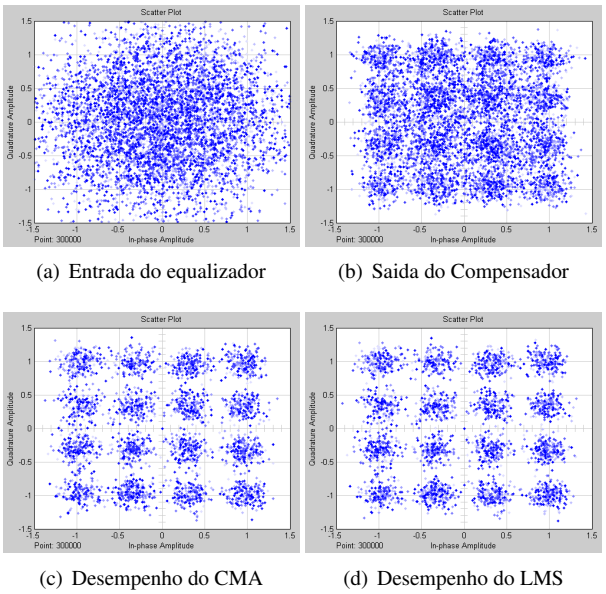
(a) Entrada do equalizador (b) Saída do Compensador

(c) Desempenho do CMA (d) Desempenho do LMS

Figura 3. Constelação 4QAM para 200 km de fibra.

convergência. Foi ainda desenvolvido um outro compensador de dispersão cromática mas para 500 km, apresentando os dois compensadores óptimos resultados, sendo possível consultar este resultado em [12].

Como se observa na figura 3, o desempenho dos algoritmos adaptativos é bastante razoável pelo que a constelação 4QAM aparece bem definida, com uma nuvem à volta de cada ponto da constelação quer para o algoritmo CMA, quer para o algoritmo LMS. Verifica-se também que os compensadores têm um papel importante pois vão aumentar o desempenho dos algoritmos.



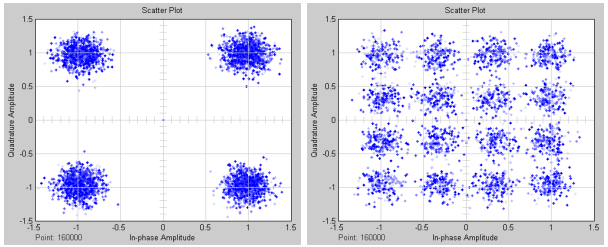
(a) Entrada do equalizador (b) Saida do Compensador

(c) Desempenho do CMA (d) Desempenho do LMS

Figura 4. Constelação 16QAM para 200 km de fibra.

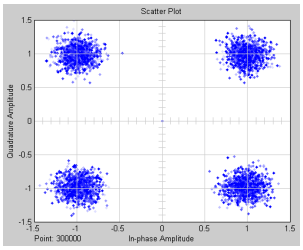
Para o caso da constelação 16QAM (figura 4) nota-se o bom desempenho dos algoritmos CMA e LMS, como no

caso anterior.

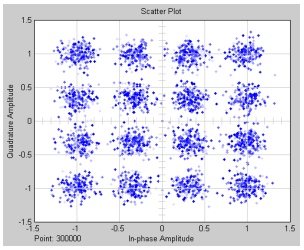


(a) Desempenho do CMA em FPGA - 4QAM

(b) Desempenho do CMA em FPGA - 16QAM



(c) Desempenho do LMS em FPGA - 4QAM



(d) Desempenho do CMA em FPGA - 16QAM

Figura 5. Resultados após co-simulação em hardware para 4QAM e 16QAM.

Posto isto, na figura 5 são apresentados os resultados da implementação dos algoritmos em hardware, usando a FPGA Virtex 4. Apresenta-se o desempenho do CMA para 4QAM (figura 5(a)) e 16QAM (figura 5(b)) onde é perceptível o bom funcionamento deste algoritmo como era de esperar, comparando com os resultados anteriores.

O mesmo se passa com o algoritmo LMS, onde a figura 5(c) mostra o resultado para a constelação 4QAM e a para 16QAM o resultado é exibido pela figura 5(d).

Verifica-se, portanto, que os algoritmos CMA e LMS apresentam bom desempenho, uma vez que as respectivas constelações se apresentam bem definidas após o seu funcionamento.

5. Conclusões e Trabalho Futuro

Este trabalho envolveu o estudo e a implementação de algoritmos adaptativos para equalização de sinal em sistemas ópticos coerentes. A implementação foi realizada em *System Generator* e implementada em hardware através de co-simulação.

Esta metodologia é extremamente recente e é possível, graças ao poder de abstracção da ferramenta de desenvolvimento *System Generator*, obter resultados em simulação e/ou transportá-los para o ambiente MATLAB, a fim de serem tratados para boa interpretação dos mesmos.

Um tipo adicional de paralelismo pode ser implementado nestes algoritmos, em que o processamento dos símbolos é realizado em paralelo, isto é, utilizando vários módulos com implementações quer do CMA, quer do LMS, consegue-se reduzir a latência de cada algoritmo. Estes podem apresentar na saída um símbolo a cada dois ciclos de relógio, reduzindo grandemente a latência de execução. Cada módulo teria os seus próprios coeficientes que seriam actualizados por cada instância colocado em paralelo.

Uma outra funcionalidade interessante seria a implementação de algoritmos *FeedForward*. Estes algoritmos poderão desempenhar um papel fundamental na sincronização da portadora óptica, aquando da presença de ruído de fase que o transmissor provoca.

Referências

[1] Sílvio A. Abrantes, *Processamento Adaptativo de Sinais*, Fundação Calouste Gulbenkian, Lisboa, 2000.

[2] S. Chen, Q. Yang, Y. Ma, and W. Shieh, *Multi-gigabit real-time coherent optical OFDM receiver*, Optical Fibre Communication/National Fibre Optic Engineers Conference (OFC/NFOEC),(OSA, 2009), Paper OTuO4, 2009.

[3] J. Machado da Silva, J. Canas Ferreira, and J. Correia Lopes, *Modelo de escrita e formatação de dissertações/relatórios de projecto do MIEEC*, Maio 2008.

[4] Micram Microelectronic GmbH, *VEGA ADC30. 30GS/s / 6-bit High-Speed Analog to Digital Converter*, 2009.

[5] D. Godard, *Self-recovering equalization and carrier tracking in two-dimensional data communication systems*, IEEE transactions on communications **28** (1980), no. 11, 1867–1875.

[6] E. Ip, A. Lau, D. Barros, and J. M. Kahn, *Coherent Detection in Optical Fiber Systems*, Optics Express **16** (2008), no. 2, 753–791.

[7] N. Kaneda, Q. Yang, X. Li, W. Shieh, and Y.K. Chen, *Realizing Real-Time Implementation of Coherent Optical OFDM Receiver with FPGAs*, Proceedings-ECOC 2009 (2009).

[8] A. Leven, N. Kaneda, and Y.K. Chen, *A real-time CMA-based 10 Gb/s polarization demultiplexing coherent receiver implemented in an FPGA*, Proceedings of the Conference on Optical Fibre Communications (OFC 2008), San Diego, CA, USA, Paper OTuG3, 24th-28th February, 2008.

[9] A. Leven, N. Kaneda, A. Klein, U.-V. Koc, and Y.-K. Chen, *Real-Time Implementation of 4.4 Gbit/s QPSK Intradyne Receiver Using Field Programmable Gate Array*, Electronics Letters **42** (2006), no. 24, 1421–1422.

[10] Xi-Lin Li and Xian-Da Zhang, *A Family of Generalized Constant Modulus Algorithms for Blind Equalization*, IEEE Transactions on Communications **54** (2006), no. 11, 1913–1917.

[11] Micram Microelectronic GmbH, *25GS/s Digital-to-Analog Converter (DAC) Demonstrator*, 2009.

[12] J. C. Ferreira N.M. Pinto, L. M. Pessoa and H. M. Salgado, *FPGA Implementation of Signal Processing Algorithms in Coherent Optical Systems*, SEON 2009 (Amadora), June 2009.

[13] L. M. Pessoa, H. M. Salgado, and I. Darwazeh, *Joint Mitigation of Optical Impairments and Phase Estimation in Coherent Optical Systems*, IEEE LEOS Summer Topical Meetings 2008 (Mexico), July 2008, Paper TuE4.3, pp. 169–170.

[14] L. M. Pessoa, H. M. Salgado, and I Darwazeh, *Performance evaluation of phase estimation algorithms in equalized coherent optical systems*, IEEE Photonics Technology Letters **17** (2009), 1181–1183.

[15] Y. Sato, *A Method of Self-Recovering Equalization for Multi-level Amplitude Modulation*, IEEE transactions on communications **23** (1975), 679–682.

[16] S. Savory, *Digital filters for coherent optical receivers*, Optics Express **16** (2008), no. 2, 804–817.

[17] R. Waegemans, S. Herbst, L. Holbein, P. Watts, P. Bayvel, C. Fürst, and R.I. Killey, *10.7 Gb/s electronic predistortion transmitter using commercial FPGAs and D/A converters implementing real-time DSP for chromatic dispersion and SPM compensation*, Optics Express **17** (2009), no. 10, 8630–8640.

[18] B. Widrow and M.E. Hoff, *Adaptive switching circuits*, (1960).

[19] Xilinx, *System Generator for DSP - Reference Guide*, Release 10.1, March 2008.

[20] Xilinx, *System Generator for DSP - User Guide*, Release 10.1, March 2008.

Reconfigurable Architectures for Next Generation Software-Defined Radio

Nelson Silva[†], Arnaldo S. R. Oliveira[†], Nuno Borges de Carvalho[‡]
[†]*DETI – IEETA,* [‡]*DETI – IT, University of Aveiro*
{*nelsonsilva, arnaldo.oliveira, nbcarvalho*} *@ua.pt*

Abstract

The necessity for better radios led a paradigm shift in favour of the Software-Defined Radio (SDR). On the other hand, enabling SDR implementations for the Next Generation Wireless Networks (NGWN) will require significantly higher performance and power efficiency than current processing architectures can provide.

In this paper, we survey reconfigurable architectures tailored for high performance Digital Signal Processing (DSP) and present a baseband processing architecture designed to shorten the gap between the achievable and the NGWN processing requirements.

1. Introduction

Motivated by the high speed processing and the low power consumption requirements, Application Specific Integrated Circuits (ASICs) are traditionally used in wireless devices for performing baseband processing. However, the proliferation of wireless protocols allied to the growing need for shorter Time-to-Market (TTM) cause the design of ASIC-based radios increasingly hard and fosters new approaches such as the Software-Defined Radio (SDR) [1].

In an SDR, the baseband processing operations are carried out by reprogrammable software or logic, operating over Digital Signal Processing (DSP) units. Compared with traditional radios, SDRs can provide several important benefits. In fact, due to the inherent high flexibility, a software update can be enough to support new standards and features. Moreover, since the same hardware can be used to enable communication over multiple wireless standards (e.g. GSM, Wi-Fi, WiMAX), it fosters interoperability with other radios as also mass IC manufacturing, which may allow reducing the cost, size and weight per chip.

Due to its advantages, SDRs are expected to be among the key technologies used in future wireless communication systems. However, SDRs require processing architectures with extreme DSP capability in order to accomplish the baseband processing operations of current networks. Moreover, it is expected that Fourth Generation (4G) wireless networks will require up to three orders of magnitude more computational capacity when compared with Third Generation (3G) networks, while maintaining a low power consumption [2]. Such performance gap must be reduced. In this sense, new processing architectures with inherent high computational capacity must be explored [3].

The remainder of this paper is organized as follows. Section 2 presents the physical layer basics of a possible 4G wireless system. Section 3 provides a small survey covering reconfigurable DSP architectures. Section 4 summarizes the proposed wireless baseband processing architecture as an innovative computing model for reducing the gap between the processing requirements of NGWN and the achievable processing capacity. At last, Section 5 presents the main conclusions and the future work.

2. 4G Physical Layer Basics

4G gained importance due to the increasing demand for wireless systems with improved mobility and data rate. The expected throughput of 100Mbps up to 1Gbps, for low and high mobility situations, respectively, requires new approaches for implementing the 4G physical layer [3].

By using transceiver arrays (see Fig. 1), it is possible to increase data rate and signal robustness, which seems to be a possible approach for implementing 4G systems.

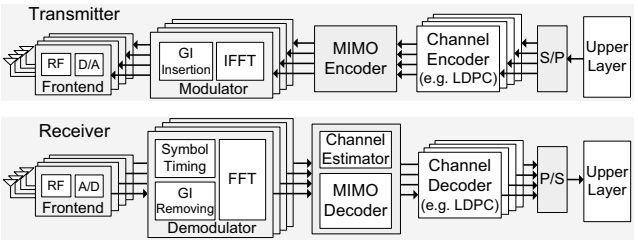


Figure 1. Example of a 4G wireless system.

In Fig. 1 it is presented the physical layer of a possible 4G wireless system [3, 4]. The major DSP-intensive blocks of the transceiver chain are the Orthogonal Frequency Division Multiplexing (OFDM) modulator/demodulator, the Multiple Input Multiple Output (MIMO) encoder/decoder and the channel encoder/decoder. The modulator converts the incoming frequency domain data into the time domain amplitude and phase signals. The demodulator converts data to the frequency domain by performing operations similar to the modulator but in the reverse order. Due to its efficient computation, the Fast Fourier Transform (FFT) algorithm is commonly used to perform the discrete time-to-frequency conversion.

The MIMO decoder is typically used for two different purposes: i) combine the received signals from the multiple antennas to generate a signal with higher robustness, which can be done through the Space Time Block Codes (STBC) algorithm, ii) multiple incoming signals are used to in-

crease the data rate, which can be achieved through the Vertical Bell Laboratories Layered Space-Time (V-BLAST) algorithm. The MIMO encoder performs the reverse operation by multiplexing data signals over multiple antennas.

The Forward Error Correction (FEC) is implemented by the channel encoder/decoder pair. Currently, high performance FEC algorithms with closer Shannon capacity are the Low Density Parity Check (LDPC) and the Turbo Code. LDPC has higher performance however, Turbo Code requires less computational capacity. Due to its superior power efficiency, LDPC and Turbo Code are expected to be among the key FEC algorithms for use in NGWN [5].

3. Reconfigurable Architectures Survey

Driven by the incessant evolution of the microelectronic technology, the development of complex integrated circuits containing several millions of transistors is now a reality. Because of that, reconfigurable devices such as Field Programmable Gate Arrays (FPGAs) have now improved performance, density and processing capacity, allowing to implement complex digital systems, eventually requiring high speed computation, temporal accuracy, memory and Input/Output (I/O) diversity.

An FPGA can be described as a matrix of configurable Logic Blocks (LBs), surrounded by I/O blocks and connected by reconfigurable interconnection resources, see Fig. 2. Due to its internal structure, FPGAs allow flexible and rapid design of complex digital systems, which may contribute for reducing TTM and Non-Recurring Engineering (NRE) costs.

Current high-end FPGAs have the equivalent capacity of millions of logic gates. These can be distributed between fixed or programmable resources, such as lookup tables (LUTs), memories, DSP macrocells, general purpose processors (e.g. PowerPC), protocol communication controllers (e.g. Ethernet) and versatile I/O blocks. Due to its attractive features, FPGAs are gaining new markets, being currently used not only for prototyping as also in commercial products, as a part of an embedded system.

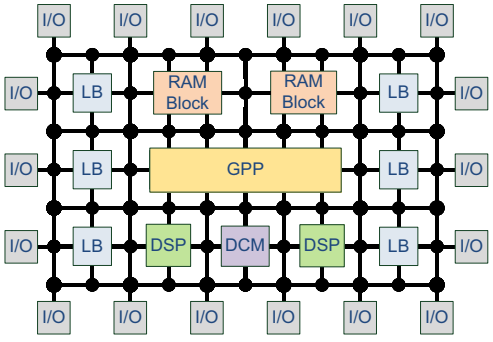


Figure 2. FPGA internal structure example.

3.1. Signal Processing on FPGA

In order to allow improved DSP, current FPGAs have DSP macrocells hardwired in its fabric, which allow higher performance and power efficiency by improving the execution of operations common to the most DSP algorithms.

DSP slices typically include a basic Arithmetic and Logic Unit (ALU), a wide multiplier, registers and interconnection logic, Fig. 3. By configuring the multiplexers it is possible to select input signals as well as to choose isolated or combined operations (e.g. an isolated multiplication or a combined multiply-accumulator operation). In addition, such interconnect flexibility not only allows an increased number of supported operations, as also it permits wide Data-Level Parallelism (DLP) by providing cascaded interconnects to other DSP slices (e.g. *PCIN* and *PCOUT*).

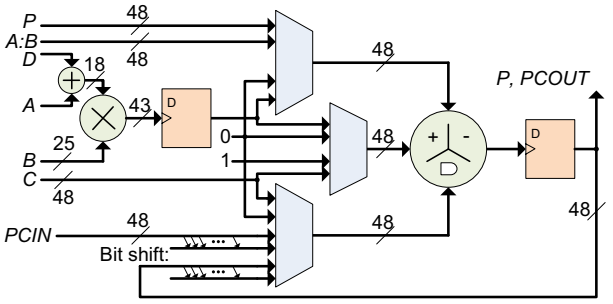


Figure 3. Simplified DSP slice, adapted from [6].

3.2. DSP Comparison With Other Devices

An exhaustive comparison between FPGAs and other IC devices is a difficult task to perform, mainly because there are many elements to compare (e.g. performance, power consumption, design flexibility, NRE costs, TTM, device size, etc.), which are not entirely dependent on the technology, some are hard to extract and all vary over the time.

However, for a certain time window, it is important to know what is the device or devices combination that is best suited for a specific application. In the NGWN context, common baseband tasks require massive signal processing under stringent power consumption constraints. Current high performance DSP architectures include ASICs, Application Specific Standard Products (ASSPs), FPGAs, DSPs or heterogeneous approaches combining two or more different processing devices. Due to its importance, these devices will be the target of the remaining discussion.

Driven by its superior performance and power efficiency, ASICs and ASSPs are commonly used in wireless devices. However, the growing number of wireless protocols make its design increasingly complex and favour the paradigm shift to the SDRs. On the other hand, DSPs allow superior design and run-time flexibility but their lower performance and power efficiency are limitative for SDR systems. FPGAs are in the middle of this tradeoff between performance, power efficiency and flexibility, see Table 1.

However, NGWN will require an estimated processing capacity increase of one to three orders of magnitude over current wireless networks while keeping a low power consumption [3]. In addition, current high performance DSP devices are already being pushed to the limit on performing the baseband processing of current wireless networks. Thereby, enabling the baseband processing of NGWN will

require a careful choose of the processing devices as also innovating processing architectures with inherent higher performance and power efficiency.

Table 1 presents a high-level comparison of several DSP implementation technologies. Although there is no perfect technology that matches with of the reprogrammability, performance and power efficiency requirements of next generation SDR, it seams reasonable to state that implementations involving DSP-enhanced FPGAs are a strong possibility.

	DSP	ASIC	ASSP	FPGA*
DSP Speed	●●○	●●●	●●●	●●◐
Power Efficiency	●◐○	●●◐	●●◐	●●○
Design Flexibility	●○○	●●●	◐○○	●●◐
Area Efficiency	●◐○	●●●	●◐○	●◐○
Reprogrammability	●●◐	◐○○	◐○○	●●○
Development Savings	●●○	◐○○	●●○	●○○
DSP Tools Support	●●●	◐○○	◐○○	●●○

Table 1. Summary comparison of DSP implementation technologies, adapted from [7, 8]. *DSP-enhanced FPGA.

Additionally, a recent study [7] comparing FPGAs with DSPs provides two important statements: 1) DSP-enhanced FPGAs allow roughly 10 to 100 times higher processing capacity than high-end DSPs; 2) FPGAs can achieve higher performance/price ratio than DSPs.

While the first achievement can be shortly explained due to the FPGAs support for massive parallel execution through a deep exploration of its resources, the second one requires a more complex explanation. In fact, since FPGAs require a considerable silicon area overhead in order to allow reconfigurability, it could be expected that FPGAs have a lower performance/price ratio over DSPs. However, it is also true that in a general purpose DSP, only a small part of the silicon is used for processing, being the major silicon area devoted for moving instructions and data around, memory and control, thus allowing to balance the performance/price ratio in favour of FPGAs.

3.3. Heterogeneous Processing Technologies

Although FPGAs represent a good solution for systems requiring very high flexibility and processing capabilities, other approaches involving different technologies can be considered. In fact, since there is no isolated technology that matches with the requirements of next generation SDR, it can be interesting to combine different technologies in order to get the advantages of each one and overcome their limitations. For example, a system composed by different processing units, such as ASICs, FPGAs and DSPs, allows to match the granularity of the hardware with the granularity of the algorithms, which in turn may result in performance and power efficiency improvements.

In [9] it is presented an heterogeneous SDR processing architecture composed by FPGAs and GPPs, all interconnected through a high-bandwidth backplane. Other approaches include picoArray PC205 [10] and XiRisc SoC [11]. While the first has a large reconfigurable array of VLIW processors and one ARM processor, the second one includes an embedded FPGA and a VLIW processor. In [12] are discussed possible heterogeneous computing problems that may appear in SDRs and it is proposed an algorithm to map the signal processing blocks of an SDR transceiver into heterogeneous processing architectures.

3.4. Multi-Processor System-on-Chip

The integration of multiple processing architectures in a single chip typically allows performance, power consumption and area benefits when compared with multiple single-chip processors. In fact, current technology is not able to achieve the NGWN processing requirements on a single processor [2], which makes vital to explore other processing approaches. Due to the known performance limitations of single processors, Multi-Processor System-on-Chip (MPSoC) are currently a common approach to build chips with higher processing capacity and allowing to follow the prediction given by the *Moore's* law [13].

The Montium tile processor [8] is an example of an energy-efficient, coarse-grained reconfigurable architecture, specially designed for DSP applications. The Montium processor is composed by five identical processing units and ten local memories, all interconnected through reconfigurable buses. Another MPSoC example tailored for SDR baseband processing includes picoArray [14], which is composed by a reconfigurable array of 430 processors, connected through a deterministic switch fabric.

4. Proposed Architecture

Achieving very high power efficiency is not a trivial task since the current technology is already being pushed to the limit. Traditionally, increasing the processor clock frequency and reducing the manufacturing technology were sufficient for meeting the market needs. However, we are reaching a boundary were increasing the clock frequency no longer scales with the computational performance as also the lithography no longer scales with the power consumption. Other approaches such as the MPSoC are now commonly used for obtaining higher performance. However, adding a high number of processors on a single chip considerably increases the complexity of the hardware, compiler, application mapping and power consumption, which is not compatible with the NGWN requirements.

On the other hand, by matching the processing architecture with the desired application, it is possible to achieve higher power efficiency, which seems to be a feasible solution for NGWN. In fact, current SDR processors are commonly MPSoC with hardware support for wide Single Instruction Multiple Data (SIMD) operations [15], which allows to significantly improve performance by exploiting the high DLP of common wireless baseband operations.

The proposed architecture goes one step forward by optimizing each Processing Unit (PU) to specific DSP kernels and by matching the interconnection between PUs with the next generation wireless baseband processing chain, Fig. 4.

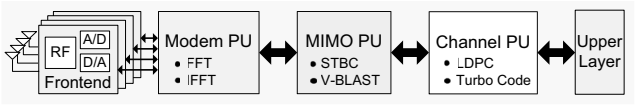


Figure 4. High level block diagram of the proposed 4G wireless physical layer processing architecture.

Contrarily to traditional MPSoC designs for SDR baseband processing, the proposed architecture avoids a global bus shared by all PUs. Instead, it was adopted a Point-to-Point (P2P) topology where each processor is connected only with its neighbours through multi-port scratchpad memory. P2P does not suffers from scaling limitations inherent to the use of a shared bus as also it allows to reduce the interconnection complexity among PUs. In addition, P2P topology permits communication parallelism as also it allows to reduce unnecessary data movement by shortening the interconnection path among consecutive PUs. By travelling shorter paths, it is possible to achieve higher throughput while minimizing the power consumption.

In order to achieve higher power efficiency, all PUs of the proposed architecture must be optimized for effective execution of the next generation wireless DSP kernels. However, dissimilar DSP kernels require different hardware solutions. For instance, a Finite Impulse Response (FIR) filter allows high DLP, being well handled by SIMD processing architectures. On the other hand, the high computational requirements of Turbo Code and its inefficient execution over software make it better handled by application specific hardware, eventually offloaded on a coprocessor [15]. Thus, achieving higher power efficiency requires specific algorithm optimization on each PU. In addition, each PU can be customized through internal processor extensions and by adding offloaded coprocessors, operating under an extended Instruction Set Architecture (ISA).

On the other hand, due to the architecture high specialization, the support for concurrent execution of dissimilar wireless protocols may lead to computational and power inefficiencies. However, an implementation based on reconfigurable hardware will provide additional flexibility by adjusting the hardware configuration to the requirements of the executing protocols.

5. Conclusion and Future Work

The above discussion summarized reconfigurable DSP architectures suitable for SDR, capable of deliver high performance by exploiting approaches involving MPSoC, SIMD and combinations with other technologies. However, the baseband processing requirements of the next generation SDR lead to a careful choose of the implementation technology as also of innovating processing architectures.

Due to the high flexibility, performance and power efficiency, FPGAs seem to be a strong technology for next generation SDR. Moreover, by matching the PUs with the NGWN physical layer, the presented architecture should be capable of achieving higher power efficiency.

Future work includes the development of a prototype based on the proposed architecture, followed by an extensive evaluation which will allow to quantify the achievements made on performance and power efficiency.

References

[1] J. Mitola, "The Software Radio Architecture," *IEEE Communications Magazine*, vol. 33, no. 5, pp. 26–38, May 1995.

[2] Mark Woh, et al., "The Next Generation Challenge for Software Defined Radio," *Lecture Notes in Computer Science*, vol. 4599, pp. 343–354, 2007.

[3] M. Woh, Y. Lin, S. Seo, T. Mudge, and S. Mahlke, "Analyzing the Scalability of SIMD for the Next Generation Software Defined Radio," in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing ICASSP*, 2008, pp. 5388–5391.

[4] H. Taoka, K. Higuchi, and M. Sawahashi, "Field Experiments on Real-Time 1-Gbps High-Speed Packet Transmission in MIMO-OFDM Broadband Packet Radio Access," in *Proc. VTC 2006-Spring Vehicular Technology Conference IEEE 63rd*, vol. 4, 2006, pp. 1812–1816.

[5] T. Lestable, et al., "Block-LDPC Codes vs Duo-Binary Turbo-Codes for European Next Generation Wireless Systems," in *Proc. VTC-2006 Fall Vehicular Technology Conference 2006 IEEE 64th*, 2006, pp. 1–5.

[6] Xilinx, *Virtex-6 FPGA DSP48E1 Slice, ug369*, Sep. 2009.

[7] Berkeley Design Technology, Inc., "BDTI Focus Report: FPGAs for DSP, Second Edition," BDTI, Tech. Rep., 2006.

[8] P. Heysters, G. Smit, and E. Molenkamp, "A Flexible and Energy-Efficient Coarse-Grained Reconfigurable Architecture for Mobile Systems," *The Journal of Supercomputing*, vol. 26, pp. 283–308, 2003.

[9] F. Van Hooft, "A Heterogeneous Software Defined Radio Architecture for Electronic Signal Interception, Identification and Jamming," in *Proc. IEEE Military Communications Conference MILCOM*, vol. 2, 2003, pp. 1178–1183 Vol.2.

[10] picoChip. (2009) PC205 High Performance Signal Processor, product brief. [Online]. Available: http://www.picochip.com/

[11] A. Cappelli, et al., "XiSystem: a XiRisc-Based SoC with a Reconfigurable IO Module," in *Proc. Digest of Technical Papers Solid-State Circuits Conference ISSCC. 2005 IEEE International*, 2005, pp. 196–593 Vol. 1.

[12] V. Marojevic, X. Reves, and A. Gelonch, "Computing Resource Management for SDR Platforms," in *Proc. IEEE 16th International Symposium on Personal, Indoor and Mobile Radio Communications PIMRC 2005*, vol. 1, September 2005, pp. 685–689.

[13] G. E. Moore, "Cramming More Components Onto Integrated Circuits," *Electronics*, vol. 38, no. 8, April 1965.

[14] D. Pulley and R. Baines, "Software Defined Baseband Processing for 3G Base Stations," in *Proc. 3G Mobile Communication Technologies 4th International Conference on (Conf. Publ. No. 494)*, 2003, pp. 123–127.

[15] Mark Woh, et al., "From SODA to Scotch: The Evolution of a Wireless Baseband Processor," in *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 152–163.

Implementation of an 128 FFT for a MB-OFDM Receiver

Bruno Fernandes
INESC-ID
bruno.jesus.fernandes@gmail.com

Helena Sarmento
INESC-ID/IST/TU
helena.sarmento@inesc.pt

Abstract

MultiBand OFDM (MB-OFDM) is a short-range wireless technology that permits data transfers at very high rates, from 53.3 Mbps to 480 Mbps. MB-OFDM uses the already licensed radio spectrum, between 3.1 GHz - 10.6 GHz, in an unlicensed manner, i.e. without a licensing cost or control. MB-OFDM divides the spectrum allocated to UWB into 14 bands of 528 MHz. Each OFDM symbol is transmitted across a band.

The FFT processor is a crucial block in multicarrier systems like OFDM, being responsible by the demodulation of the OFDM symbol. In this paper we analyze the different FFT architectures provided by the Xilinx FFT CORE Generator and propose the design of a 128-point Pipeline FFT to implement the OFDM demodulation at the receiver.

1. Introduction

UWB is a low power, short range and high speed wireless technology for wireless personal area networks. Potential indoor applications for UWB are high speed multimedia streaming connections between devices, such as digital video recorders, set-top boxes, televisions and PC peripherals. FCC in USA and the European Commission authorize the use of UWB, for communication systems, in 3.1 GHz - 10.6 GHz band.

MB-OFDM [1] is an OFDM specification for UWB. MB-OFDM divides the available spectrum (3.1 GHz - 10.6 GHz) into 14 sub-bands, each one occupying 528 MHz. Each OFDM symbol, combining a set of 128 narrow band sub-carriers, is transmitted in one of the bands. A total of 100 data sub-carriers and 10 guard sub-carriers are used per symbol. In addition, 12 pilot sub-carriers allow for coherent detection. The timing parameters associated with the OFDM PHY are listed in Table 1. The time to process an OFDM symbol is 242.42 ns (1/4.125 MHz).

OFDM modulation and demodulation are implemented by IFFT and FFT processors. These modules have high computational complexity. Therefore, due to time constraints, the FFT block is the more critical module in the hardware implementation of the physical layer, at the receiver.

MB-OFDM receivers are typically implemented with ASICs. However, the latest generations of FPGAs, including DSP capabilities, embedded processors and special features for I/O streaming are powerful enough that they encourage the design of wireless applications using FPGAs. We are currently implementing the physical layer of a MB-OFDM receiver using FPGAs[1].

Table 1. Timing-related Parameters

Sampling frequency	528 MHz
Total number of sub-carriers (FFT size)	128
Number of data sub-carriers	100
Number of pilot sub-carriers	12
Number of guard sub-carriers	10
Number of null sub-carriers	0
Sub-carrier frequency spacing (Δf)	4.125 MHz
IFFT and FFT period	242.42 ns

This paper discusses the use of architectures provided by the Xilinx FFT IP Core generator [2] to implement the OFDM demodulation process in the MB-OFDM receiver. Two architectures were implemented in order to achieve the desired performance (242.42 ns).

The paper is organized as follows. Section 2 introduces the characteristics of MB-OFDM symbol and FFT algorithm concepts. In Section 3, we analize the potentialities of the Xilinx FFT Core Generator, describing its architectures and the resources used in each one. The different implementations for the OFDM receiver are described in Section 4. Section 5 presents the results obtained for the different implementations. Finally, conclusions and future work are discussed in Section 6.

2. The MB-OFDM Symbol

OFDM is a transmission scheme that combines multiplexing and modulation. Data symbols are split up into a set of independent smaller symbols (sub-symbols). Each sub-symbol is modulated on a separate sub-carrier (FDM), but the set of sub-symbols define the OFDM symbol. Orthogonality between sub-carriers permits overlapping of sub-carriers spectrum without mutual interference. Thus, high spectral effi-

[1]UWB Receiver: baseband processing using reconfigurable hardware, *Project Ref: PTDC/EEAELC/67993/2006*, Funded by Fundação para a Ciência e Tecnologia.

ciency is obtained.

$$s_n(t) = \begin{cases} \sum_{n=-\frac{N}{2}}^{n=\frac{N}{2}} C_n e^{j2n\pi\Delta_f t} & 0 \le t \le T \\ 0 & t < 0 \text{ or } t > T \end{cases} \quad (1)$$

Mathematically, the OFDM symbol is expressed as a sum of pulses shifted in time and frequency and multiplied by the data symbols (equation 1) [3]. The C_n coefficients represent data (QPSK or DCM subsymbols), pilot or training symbols that are transmitted simultaneously at the N subcarriers spaced by Δf. Time duration of symbol is $T = \frac{1}{\Delta f}$ (Table 1). The complex notation of (1) is, in fact, nothing more than the inverse discrete Fourier transform of the N C_n symbols. Therefore, at the receiver, the C_n symbols can be obtained implementing a Fast Fourier Transform (FFT).

2.1. FFT

The FFT is an efficient algorithm for computing the Discrete Fourier Transform (DFT). It is based on the fundamental principle of decomposing the computation of the DFT of a sequence of length N into successively smaller DFTs. The DFT $X(k)$ (k = 0,...,N-1) of a sequence $x(n) = 0, \ldots, N-1$ is defined in equation 2 as:

$$X(k) = \sum_{n=0}^{N-1} x[n] W_N^{nk} \quad (2)$$

where W_N^{nk}, referred as *twiddle factor*, is given by equation 3.

$$W_N^{nk} = e^{-jnk\ \frac{-2\pi}{N}} \quad (3)$$

From the definition in [4], algorithms which decompose the sequence $x[n]$ into successively smaller subsequences are called decimation-in-time algorithms, while decimation-in-frequency algorithms decompose the output $X(k)$.

The Radix-2 algorithm separates the FFT computation of $x[k]$ in even and odd numbered points. Taking advantage of the properties of the twiddle factor, equation (2) can be rewritten as equation (4).

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x[2r] W_{N/2}^{2rk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x[2r+1] W_{N/2}^{2rk} \quad (4)$$

Computation of the original DFT is now reduced to two DFT with half the complexity. This procedure can be recursively applied in order to reduce the number of arithmetic operations needed. In a Radix-4 algorithm, the same method is applied to a power of 4 data sequence N, dividing the sequence by 4 in order to let all the elementary computations be done by 4-point DFTs.

3. XILINX FFT (Core) Generator

Xilinx FFT IP Core [2] follows the Cooley-Tukey algorithm [4] to calculate the FFT. The FFT core can compute any N-point forward or inverse DFT (IDFT) as long as $N = 2^m$ with $m = 3, \ldots, 16$. Xilinx FFT IP Core implements Radix-4 and Radix-2 decomposition for computing the DFT .

The Core provides four different optional architectures: Pipeline, Streaming I/O; Radix-4 Burst I/O; Radix-2 Burst I/O and the Radix-2 Lite Burst I/O. These different options present a trade-off between core size and transform time (see Figure 1).

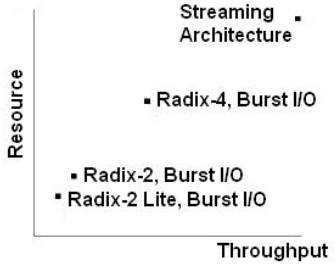


Figure 1. Resource Vs. Throughput

The following features are common to all architectures: memory to store twiddle factors can be block Ram or distributed RAM; input data is separated in real and imaginary; input data is presented in natural order in two's complement or single precision floating point format; output data can be configured to be presented in natural or inverted order.

Three arithmetic options are available for computing the FFT: full-precision unscaled, scaled fixed-point and block floating-point. When using full-precision unscaled arithmetic, the data path increases, retaining all integer bits. Fractional bits are truncated or rounded after the multiplication on the butterflies. The number of bits of output data is given by $X + \log_2(Y) + 1$ where X is the input width and Y is the transform length. In scaled fixed-point, the user configures the scaling in each stage of the FFT. The scaling parameters can be asserted in real-time. In block floating-point, scaling is performed in run-time by the core to prevent data overflow.

3.1. Radix-4 and Radix-2, Burst I/O

In all Burst I/O architectures, the Decimation-In-Time method is used. The difference between both solutions relies on the butterfly processing engine used in each structure. The Radix-4, Burst I/O solution uses radix-4 butterfly processing engine (Figure 2) for FFT computation while the Radix-2, Burst I/O uses radix-2 butterfly processing engine (Figure 3). For both, the transform is calculated after the full frame is loaded and the output data is unloaded after the computation is finished. Overlap of loading and unloading of data is possible if the output is in bit reversed order.

978-972-789-304-1 REC'2010

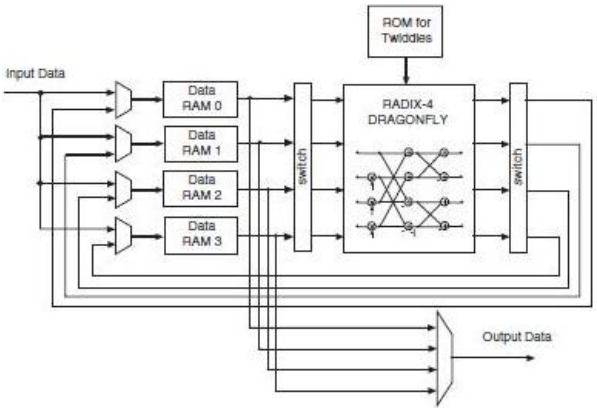


Figure 2. Radix-4, Burst I/O [2]

In Radix-4, the architecture consists of $\log_4 N$ stages when used for N-point sizes power of 4, with each stage containing $\frac{N}{4}$ Radix-4 butterflies. An extra Radix-2 stage is used for combining data when the point size is not a power of 4. In a Radix-2, each of the $\log_2 N$ stages contain $\frac{N}{2}$ butterflies.

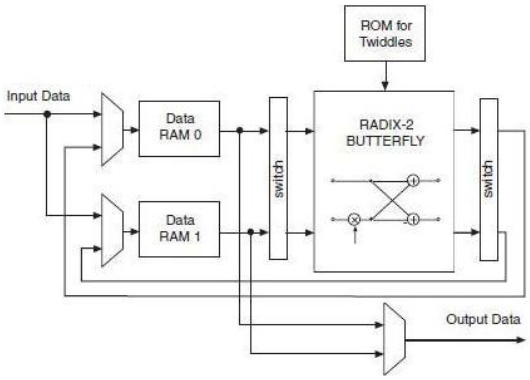


Figure 3. Radix-2, Burst I/O [2]

3.2. Radix-2 Lite, Burst I/O

This architecture uses one radix-2 butterfly to compute the DFT, but shares one adder/subtractor. Hardware resources are reduced at the expense of an additional delay per butterfly calculation. Since the additions and subtractions are performed by the same block, the Radix-2 butterfly outputs one value at a time (Figure 4). Real and imaginary multiplications are done independently, as such only one twiddle factor is read from the ROM memory at each clock cycle. Data can only be simultaneously loaded and unloaded if the output samples are in bit reversed order.

3.3. Pipeline, Streaming I/O

The Pipelined, Streaming I/O architecture is implemented using the Decimation-In-Frequency (DIF). Several pipelined Radix-2 butterflies allow continuous data processing (Figure 5). Each butterfly has its own

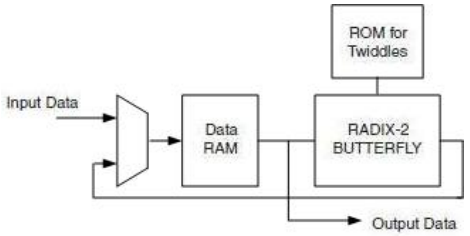


Figure 4. Radix-2 Lite, Burst I/O

memory banks to store input and intermediate data.

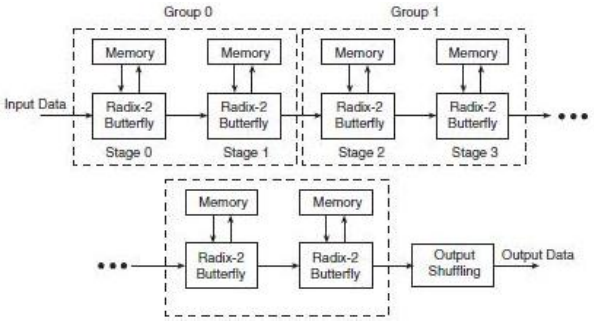


Figure 5. Pipelined, Streaming I/O [2]

This architecture has the ability to simultaneously perform transform calculations on the current frame of data, load input data from the next frame of data, and unload the results of the previous frame of data. The design also allows computation of frames with gaps in between.

4. FFT Implementation

The Pipeline, Streaming I/O architecture allows continuous data processing, making it the most suitable for real-time or wireless applications where input data arrive serially. This architecture also offers the highest throughput making it the more appropriate for high rate wireless applications. Therefore, we used it as the base design to implement the OFDM demodulator of the MB-OFDM Receiver.

We firstly designed a 128-point FFT, using a Pipeline, Streaming I/O architecture. However, in a Virtex-4, it didn't fulfill the 242.42 ns processing time requirement. Therefore, we analyzed the parallelization of the FFT processing.

For all implementations, we assumed that the input stream was previously written into memory blocks and that its format is according to the specification of Xilinx core. Also, all implementations use full-precision unscaled arithmetic, with an 8 bit input and output. The final values preserve the input fractional bits .

In order to improve the performance, we decide to use two 64-point FFT (FFT2pipe). In the FFT2pipe, the input is parallelized so that even and odd data are computed separately. Final values are calculated with a radix-2 block (Figure 6). The smaller FFTs

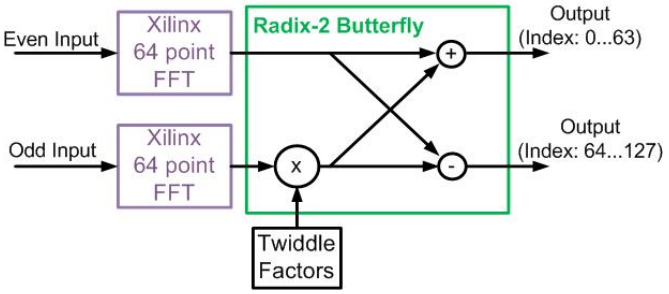


Figure 6. Architecture with two 64-point Xilinx FFTs

are generated by the FFT core Generator and also have a Pipeline, Streaming I/O Architecture. Real and imaginary twiddle factors are stored separately in RAM blocks. The arithmetic functions implement in the radix-2 block use the Complex Multiplier and Adder/Subtracter functions for the Xilinx Core Generator, since they are optimized for Xilinx FPGAs.

For the FFT2pipe, the clock frequency needed is 264 MHz, since the 128 points input is parallelized in two subsequences. The architecture must maintain continuous data processing to be suitable for real-time or wireless applications as previously mentioned.

Due the constraints imposed by the sub-carriers demodulator (QPSK/DCM demodulator), which processes output data from the FFT processor, we redesigned the FFT2Pipe with two clock frequencies. Output data from the FFT is sent to demodulator at twice the FFT processing clock. For that, we use the DCM (Digital Clock Manager) primitive to generate the clocks.

5. Results

The architectures were synthesized, implemented and floor planned for the Virtex-4 xc4sx35 with speed rate of -10 FPGA used in the project. Test bench waveforms were created for each implementation. Table 2 presents values for the maximum clock frequency and resources. The maximum clock frequency is the best case achievable of setup clock in the place and route report.

	FFTpipe	FFT2pipe	FFT2pipe with DCM
Freq. (MHz)	278.086	253.871	293.6
Slices	1.386	1.875	2.188
4 input LUT	1.096	2.469	3.085
FIFO/RAMs	4	12	6
DSP48s	14	22	43
DCM_AVs	-	-	1

Table 2. Post layout Results

With a 128-point FFT Pipeline, Streaming I/O Architecture the required frequency to demodulate an OFDM symbol is 528 MHz. As presented on Table 2 this requirement is not achieve. By parallelizing the FFT computation, the data input must be read at a clock frequency of 264 MHz. The frequency obtained was 253 MHz. Results for the FFT2pipe with DCM primitive are also presented. For this implementation, the FFT clock frequency is half the maximum frequency of the block. Maximum clock frequency is 293 MHz. Therefore the FFT maximum frequency to is 147 MHz.

6. Conclusions and Future Work

This paper presents the implementation of a 128-point FFT for the OFDM demodulation in a MB-OFDM receiver. The Pipelined, Streaming I/O architecture, provided by the core generator from Xilinx, is used.

Results show that the 128-point FFT can not achieve time requirements for Virtex-4 devices. However, with two 64 point FFT in parallel, the maximum clock frequency is very close to the identified requirements. Post layout results demonstrated that for a similar Virtex-4 device with speed rate of -11, timing requirements are satisfied. An implementation, using a DCM primitive is presented, but the need to have a clock frequency twice the FFT clock frequency is no longer need.

To completely fulfill the requirements, further parallelization of the input will be implemented. The same methodology will be applied. An architecture using four 32-point FFTs and a radix-4 block, maintaining continuous data processing, will be implemented.

We believe that the final design will be appropriate for a MB-OFDM Receiver on the Virtex-4 FPGA.

References

[1] ECMA. *Standard ECMA-368: High Rate Ultra Wideband PHY and MAC Standard*, December 2008.

[2] Xilinx Inc. *Logic Core Fast Fourier Transform v6.0, DS260*, September 2008.

[3] Nuno Rodrigues, Horácio Neto, and Helena Sarmento. A OFDM module for a MB-OFDM Receiver. *Design & Tecnology of Integrated Systems*, 2007.

[4] J.W.Cooley and J.W.Tukey. An algorithm for machine computation of complex fourier series. *Math Comput*, 1965.

[5] Federal Communications Commission. *Revision of Part 15 of the Commission Rules Regarding Ultra-Wideband Transmission Systems*, Report FCC 02-48, February 14, 2002.

Sessão Regular 3

Aplicações Científicas

Moderação: Mários Véstias
Instituto Superior de Engenharia de Lisboa / INESC-ID

Validação e Concretização do Módulo MICTP do Primeiro Nível do Filtro de Eventos do Detector ATLAS

Bruno Fernandes
FCUL-DF
bruno.jesus.fernandes@gmail.com

Per Klöfver
CERN

Ralf Spiwoks
CERN
Ralf.Spiwoks@cern.ch

Guiomar Evans
FCUL-DF
gevans@fc.ul.pt

Stefan Haas
CERN
Stefan.Haas@cern.ch

José Soares Augusto
FCUL-DF
jasa@fisica.fc.ul.pt

Resumo

O projecto da miríade de subsistemas que integram o detector ATLAS, do LHC[1] *do CERN, espraiaram-se por cerca de uma dúzia de anos, o que fez com que muitas das tecnologias inicialmente utilizadas, nomeadamente aquelas associadas à electrónica, viessem a padecer de obsolescência. Tal é o caso do antecessor do módulo MICTP aqui descrito, que foi inicialmente concebido para ser implementado em vários CPLDs [1].*

No último par de anos, com o LHC prestes a entrar em funcionamento e com uma mole humana de cientistas já a planear aquele que virá a ser denominado SLHC (o 'S' é de Super), muita electrónica tem vindo a ser re-concebida e re-projectada, por forma a diminuir a complexidade da actual implementação e, nalguns casos, para aumentar o leque de funções disponíveis mas, acima de tudo, para preparar os sistemas para se encaixarem naturalmente no futuro SLHC.

Neste documento são apresentadas a concretização e a validação da nova incarnação do MICTP, um módulo integrante do primeiro nível de trigger *(denominado LVL1 ou FLT*[2]*) do ATLAS. O módulo foi descrito em VHDL e a sua validação assentou nas simulações funcional e temporal e nos resultados da síntese, quer no que respeita aos recursos ocupados na FPGA, quer no que respeita à satisfação das especificações temporais. O MICTP foi testado em laboratório, tendo sido validados o funcionamento intrínseco dos vários blocos e a comunicação com os outros módulos do FLT com os quais o MICTP faz fronteira.*

Dá-se ênfase ao conjunto de ferramentas integradas no sistema de desenvolvimento utilizado na concretização do projecto e ao fluxo de projecto. O módulo foi implantado numa FPGA Stratix II EP2S60 *da* Altera.

[1]LHC – Large Hadron Collider; ATLAS – A Toroidal LHC ApparatuS.

[2]LVL1 – "Level 1"; FLT – "First Level Trigger".

1. Introdução

O maior projecto do CERN é o acelerador LHC, construído num túnel circular com 27 Km de perímetro, enterrado a 100 metros de profundidade, onde colidem protões com velocidades muito elevadas. À máxima velocidade prevista, cerca de $0.999999991 \times c$, (onde c é a velocidade da luz no vácuo) cada protão tem uma energia cinética de 7 TeV. Esta energia, altamente concentrada, permite gerar um conjunto elevado de partículas nas colisões entre enxames de protões. Espera-se que, ocasionalmente, uma das partículas geradas seja o bosão de Higgs – se ele, de facto, existir. Em alguns pontos da circunferência que constitui o LHC os tubos intersectam-se para que as partículas possam colidir. É nestes pontos que se encontram os detectores (também denominados *experiências*) ATLAS (Figura 1), CMS, ALICE e LHCb.

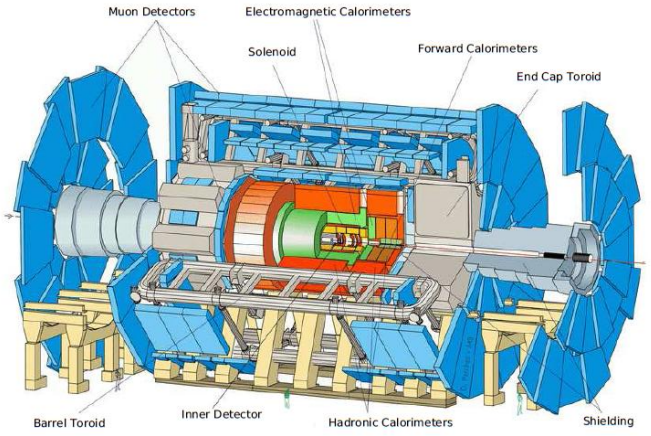


Figura 1. Esquema do detector ATLAS.[2]

Os protões viajam agrupados em enxames – os *bunches*. Existem 2808 *bunches* a viajar em simultâneo à volta do LHC, separados entre si de 25 ns, que se intersectam a uma frequência de 40 MHz. A maioria dos protões não colide: prevê-se que em cada interacção entre feixes, i.e., em cada *Bunch Crossing* (BC),

existam em média 23 colisões protão-protão.

Quase todas as colisões irão dar lugar a processos Físicos banais, e por isso não se justifica guardar toda a informação registada em cada BC. À taxa de colisões de 40 MHz é impossível armazenar todos os dados, uma vez que a capacidade de encaixe estimada como necessária rondaria os 80 TBytes/s. Foi implementado um filtro de acontecimentos, denominado *trigger* do ATLAS, para registar apenas dados de processos raros, interessantes ou originais, como será o bosão de Higgs, que se estima poder ser observado apenas uma vez em cada 10^{13} BCs. O conjunto de dados de cada BC é referido como um *evento*.

O sistema de *trigger* possui três níveis. O primeiro, o *LVL1* é implementado em *hardware* e reduz a taxa de eventos de 40 MHz para cerca de 75 KHz. O nível seguinte, o *LVL2*, redu-la para cerca de 1 a 2 KHz, enquanto que o último nível, o *Event Filter*, finaliza o processo reduzindo a frequência de eventos para cerca de 100 a 200 Hz (Figura 2). Apesar desta selecção de dados, a taxa na saída do filtro de eventos é cerca de 300 MBytes/s. O sistema identifica a que BC pertence cada evento registado, o que não é trivial pois quando as partículas criadas num dado BC cruzam os detectores periféricos já entretanto teve lugar o BC seguinte no núcleo do ATLAS.

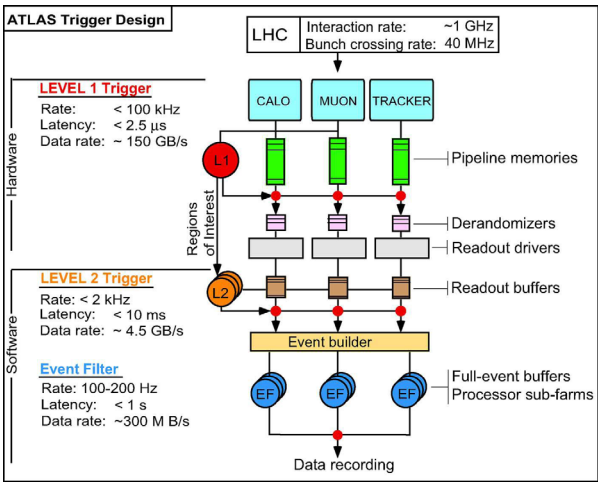


Figura 2. O sistema de *trigger* do ATLAS.[2]

Neste documento é descrito o re-projecto e implementação do módulo MICTP, parte integrante do *trigger* de Muões do nível LVL1 do *trigger*. Na secção 2 são descritos o LVL1 e o sistema MUCTPI ("Muon-to-CTP Interface") onde o MICTP se integra. As especificações temporais e funcionais do módulo são descritas na secção 3. Os resultados das simulações e da síntese, e os testes de validação em laboratório, são apresentados em 4. Finalmente, as conclusões relativas ao trabalho desenvolvido são expostas na secção 5.

2. O nível LVL1 e o MUCTPI

A função principal do LVL1 é emitir um sinal que indica se o BC deve ser retido para análise futura por, potencialmente, apresentar interesse para a Física [2].

O valor máximo da latência do LVL1 ronda os 2 a 3 μs. Durante este intervalo de tempo toda a informação proveniente dos detectores deverá permanecer armazenada em *buffers*. Estes dados são retidos apenas quando o LVL1 activa o sinal de "Nível 1 Aceite", *L1A – Level 1 Accept*: quando isso não acontece a informação associada à totalidade do evento é descartada. A actual taxa máxima de leitura no LVL1 é de 75 KHz, um majorante do fluxo de informação que flui para o nível seguinte (LVL2).

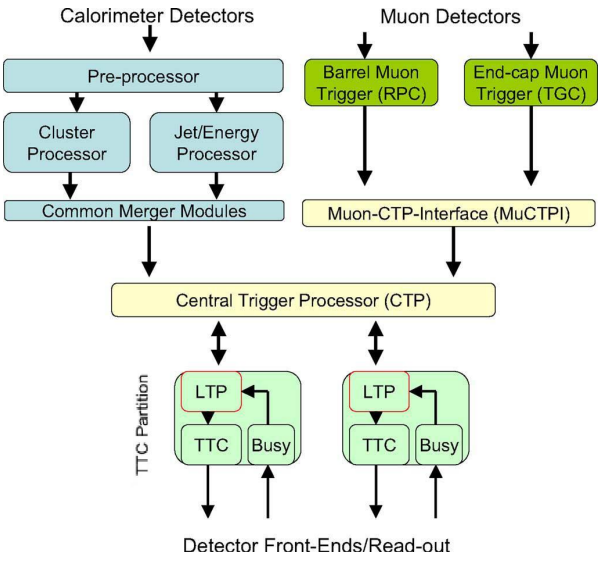


Figura 3. O nível LVL1.[2]

O LVL1 consiste do *trigger* de Calorímetro, do *trigger* de Muões e do Processador Central de *Trigger*, denominado *CTP* (Figura 3). Devido à grande quantidade de detectores, o LVL1 monitoriza 208 sectores. Cada um deles pode enviar no máximo duas trajectórias de muões, os *muões candidatos*, para o MUCTPI. Os dados associados a estes muões descrevem a sua posição e momento transversal, denominado p_t. É o valor destas variáveis que dá uma indicação do seu "interesse para a Física" e que, portanto, irá fazer com que o evento seja registado ou descartado.

O MUCTPI soma o número de candidatos em cada uma das 6 categorias pré-estabelecidas (ou intervalos de p_t). Estas somas, denominadas *multiplicidades*, são enviadas para o CTP em cada BC. Quando o L1A é activado, devido "à suspeita" de um elevado nível de interesse no evento, o MUCTPI envia os dados dos muões candidatos no presente BC para os níveis superiores do filtro de eventos.

O MUCTPI é constituído pelos 16 MIOCTs ("Octant Modules"), pelos módulos MICTP e MIROD, e pelo *backplane* MIBAK (Figura 4). Está também equipado com uma placa de interface com um com-

978-972-789-304-1 REC'2010

putador que é usada para configuração do sistema.

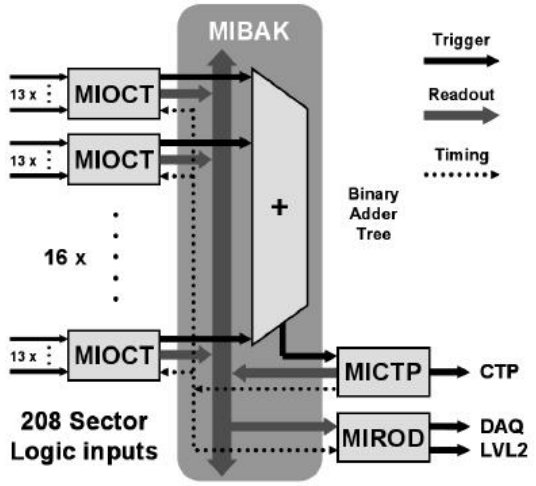


Figura 4. Arquitectura do MUCTPI.[1]

Os 16 MIOCTs recebem a informação referente aos muões candidatos dos sectores lógicos das câmaras RPC e TGC[3], sincronizam os sinais, asseguram que nenhum dos muões candidatos é contado duas vezes (o que pode acontecer devido à sobreposição física - *overlap* - dos detectores) e soma as multiplicidades para cada categoria de p_t. O total das multiplicidades é calculado por somadores realizados com FPGAs situadas na MIBAK, sendo em seguida enviado para o CTP.

No CTP é activado ou não o sinal L1A, consoante haja ou não relevância nos dados. No caso afirmativo este sinal é enviado para o MICTP e, depois, para os MIOCTs através do MIBAK. Este procedimento irá iniciar a leitura de todos os dados associados aos muões candidatos do BC correspondente a partir dos MIOCTs, e das multiplicidades a partir do MICTP, dados estes que são enviados para o MIROD pelo *bus* de dados existente no MIBAK. O MIROD envia os dados de até 16 candidatos para o LVL2, e envia a informação desses candidatos para o sistema DAQ (*Data Aquisition*), o responsável pelo armazenamento de todos os dados sempre que o *trigger* aceita o evento. Os dados ficam disponíveis para serem analisados pela comunidade científica.

Na FPGA implantada na placa principal do MUCTPI irão alojar-se os módulos MIROD e MICTP. Estes dois módulos são objecto de novos projecto e implementação, pelas razões já anteriormente expostas e, mais concretamente, o relato deste renascer do MICTP é feito na nossa contribuição.

3. As Especificações do MICTP

A fase inicial do re-projecto do MICTP foi relatada em [3]. Agora vai-se proceder a uma descrição a posteriori da tarefa, pois o MIOCT já foi integralmente projectado, verificado, implementado e testado, inclusive em *hardware*, num laboratório do CERN[4]. Nesta secção vai-se sucintamente descrever as funções dos vários módulos do MICTP, cuja arquitectura é mostrada na figura 5.

O conjunto de funções atribuídas ao MICTP, inclui a recepção e a formatação dos sinais provenientes do CTP e do LHC e a sua subsequente redistribuição por todo o sistema MUCTPI, o envio dos valores de multiplicidade dos muões candidatos provenientes dos 16 MIOCTs e, ainda, a monitorização do bom funcionamento de todo o módulo. As funções do MICTP são:

- receber, a cada 25 ns, os valores finais das multiplicidades dos muões e reenviá-los, preformatados, para o módulo CTP. As multiplicidades devem ser guardadas em memórias exteriores ao MICTP durante a latência do sinal L1A. Quando o L1A é activado, os valores das multiplicidades são lidos das memórias e ficam disponíveis, guardadas em FIFOs[5], durante ± 3 BCs relativamente ao BC que está a ser processado. Estes dados são posteriormente formatados e enviados para a FIFO de *Readout* até serem lidos pelo módulo MIROD;
- implementar os contadores de *reset* de eventos (ECR), de *bunches* (BCR) e do sinal L1A, e guardar os parâmetros associados ao tipo de *trigger* especificado pelo menu de *trigger*;
- receber o sinal TST que inicia o modo de teste;
- receber os sinais de relógio provenientes do circuito *delay25* e os sinais globais BCi (o relógio mestre do acelerador) e ORBIT (que faz um *reset* global na experiência LHC);
- enviar os sinais ECR, L1A, TST, BC, BCR (relacionado com o ORBIT) e MFE (que activa a monitorização), após estes terem sofrido alterações na temporização (atraso e/ou duração) para o MIBAK que, por sua vez, os reenvia para todo o MUCTPI (incluindo o MICTP).

3.1. Especificações Temporais

A interface MUCTPI deverá apresentar uma baixa latência na propagação de sinais e, com esse objectivo, cada módulo do MICTP está sincronizado com diferentes sinais de relógio gerados pelo circuito *delay25* [4]. Este possui 5 saídas e tem uma funcionalidade similar à de uma PLL: a partir do sinal de relógio BCKDI (que o MUCTPI recebe no painel frontal), gera 4 sinais de relógio (BCKD_j, com j=0,1,2,3, ver tabela 1) com igual frequência mas desfasados entre si. Uma das saídas do *delay25* emite uma réplica do BCKDI. Os atrasos em cada saída são programados individualmente.

[3]RPC – "Resistive Plate Chamber"; TGC – "Thin Gap Chambers".

[4]O primeiro autor desta contribuição efectuou esta validação em laboratório durante uma estadia de 2 meses e meio no CERN, em Genebra.

[5]As "FIFOs" são, é claro, memórias, ou registos, do tipo "First-In First-Out".

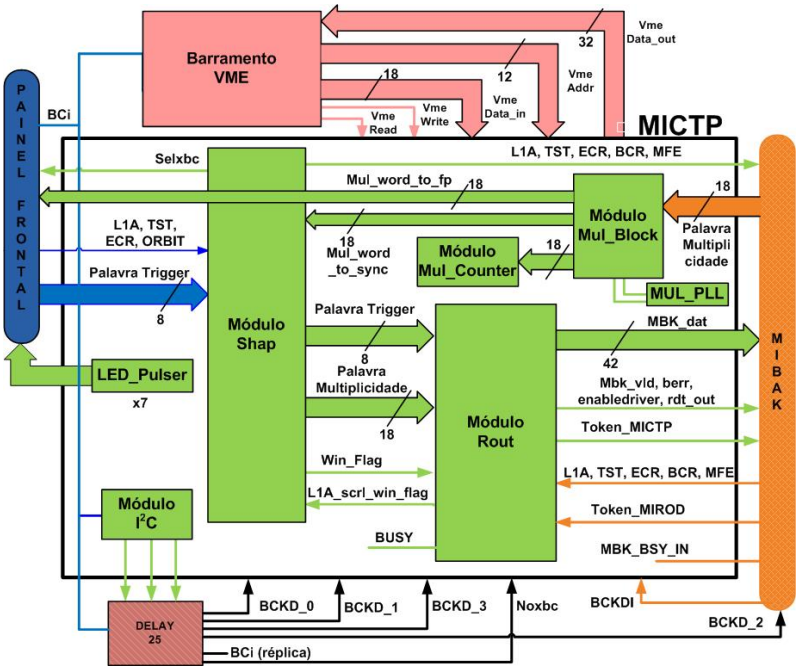


Figura 5. Diagrama de Blocos do MICTP.

Optimizando-se esta geração é possível minimizar o tempo de propagação. O MICTP possui controladores de barramentos I^2C [5], que é usado para configurar os atrasos das saídas do dispositivo *delay25*, e VME.

Relógio	Módulo
BCKD_0	*Shap*
BCKD_1	MIBAK
BCKD_2	*Mul* e *Mon_count*
BCKD_3	Relógio Interno do MICTP

Tabela 1. Domínio de cada relógio usado no MUCTPI.

A saída BCKD_3 está disponível no painel frontal, sendo referida como *BCo*. O sinal de relógio BCKD_1 é reenviado pelo MIBAK para o MICTP, sendo aqui referido como BCKI. É usado também nos módulos *Shap* e *Rout*.

3.2. Módulos do MICTP

Os módulos descritos de seguida implementam parcelarmente as funções do MICTP. Cada um possui parâmetros definidos pelo utilizador que, juntamente com outros gerados automaticamente pelos módulos, são guardados em registos acedidos pelo bus VME[6]. Na figura 5 apresenta-se um diagrama de blocos do MICTP.

Como foi já referido, cada módulo do MICTP utiliza um sinal de relógio diferente. Esta implementação possibilita a ocorrência de meta-estabilidade nos sinais partilhados por vários módulos.

Este problema é resolvido pelo bloco *SS_Synch*, cujos diagramas lógico e temporal se encontram na figura 6. Ele recebe o sinal assíncrono e está sincronizado pelo relógio do módulo alvo. Apesar de aumentar a latência do sinal, este bloco permite que o valor meta-estável (referido como META na figura 6) se resolva na primeira báscula antes de ser transmitido para o módulo. Esta implementação não garante uma resolução exacta, mas a baixa frequência dos relógios (40 MHz) e a possibilidade de optimizar a fase mútua reduz significativamente a possibilidade de ocorrerem problemas na propagação de sinais.

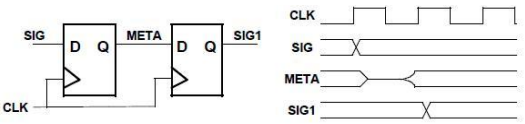


Figura 6. Diagrama Lógico e temporal do bloco SS_synch cuja função é prevenir problemas de meta-estabilidade.

3.2.1. O Módulo Shap

Este módulo efectua a formatação temporal ("shaping") de diversos sinais. Sincroniza os sinais L1A, ECR, ORBIT e TST com o relógio BCKD_0, e sincroniza as palavras de Trigger, recebida no painel frontal, e de Multiplicidade, escolhida no bloco Mul_Block, com o relógio BCKDI. É possível escolher o flanco de sincronismo do relógio.

As palavras de *Trigger* e multiplicidade são enviadas para o módulo Rout, enquanto que os sinais L1A, ECR, ORBIT e TST sofrem modificações temporais no módulo Shap. A partir destes sinais são gerados e enviados para o MIBAK os sinais MFE, BCR e Win_flag, que são então difundidos para todo o MUCTPI.

3.2.2. O Módulo Mul_Block

O Módulo Mul_block é o responsável pelas palavras de multiplicidade enviadas para a formação dos eventos e para o CTP. Elas podem ser provenientes do MIBAK ou, então, na realização de testes podem ser lidas da RAM Multi_Ram. As palavras seleccionadas são remetidas para o CTP, pelo painel frontal, e para os módulos *Shap* e Mon_Counter. Para diminuir a latência no envio para o CTP, o módulo MICTP possui uma PLL que gera um relógio idêntico ao BCKD_2 mas com um atraso de $\frac{1}{4}$ de período. A palavra de multiplicidade enviada para o CTP é síncrona com o BCKD_2. As palavras enviadas para o módulo Shap são previamente alinhadas com o sinal L1A por um *pipeline* com atraso programável de 1 a 32 períodos de relógio.

3.2.3. O Módulo Rout

O Módulo Rout é o responsável pela difusão dos dados (*readout*) e pela monitorização das multiplicidades vindas do CTP e dos dados de *trigger* vindos do LHC.

Este módulo recebe os sinais do MIBAK, previamente tratados pelo módulo Shap, assim como as palavras de *trigger* do CTP e os valores da multiplicidade escolhidos no módulo Mul_Block. Em caso de recepção de um sinal L1A, as multiplicidades obtidas antes, depois e durante o actual sinal L1A, são escritas numa memória FIFO durante uma janela de tempo definida pelo utilizador. Os números de identificação de L1A e BC, os dados de *trigger* e ainda outros dados, são guardados noutra memória FIFO. Quando ambas as memórias FIFO contêm dados novos, forma-se um evento para ser escrito nas FIFOs Readout e Monitoring. Nesta última os dados do evento podem ser consultados através do barramento VME, enquanto que os dados na FIFO Readout são transferidos para o MIROD via MIBAK.

O Rout activa o sinal BUSY quando a FIFO que guarda as multiplicidades está quase cheia, o que indica ao CTP que não deve gerar mais sinais L1A visto os dados anteriores ainda estarem a ser processados.

3.2.4. Módulo Mon_Counter

O módulo Mon_Counter é utilizado para a monitorização das palavras de multiplicidade escolhidas no bloco Mul_Block, contando o número de vezes que um dado valor ocorreu nas multiplicidades.

Cada palavra contém 6 valores de p_t, e cada um deles vale de 1 a 7. Assim, este módulo inclui 42 contadores de 16 bits (15 de contagem e 1 para a detecção de *overflows*) que registam ao longo de toda a experiência a estatística dos momentos transversais.

Após termos descrito sumária e concisamente os módulos constituintes do MICTP, passamos à descrição dos aspectos relacionados com os seus projecto e implementação.

4. Validação do MICTP

Nesta secção são apresentados os resultados da síntese e das simulações, e os procedimentos de teste em laboratório do *firmware* do MICTP.

São pertinentes alguns comentários sobre o desenvolvimento do código, tendo em conta que o MICTP se integra num sistema muito complexo, com vários módulos-fronteira já implantados há bastante tempo e cujas especificações estão congeladas, mas com outros que poderão vir a ser alterados. O código VHDL do MICTP poderá sofrer modificações no futuro e, por isso, está muito bem comentado e estruturado, de acordo com a prática interna do CERN. Muito parâmetros são configuráveis (e.g. as profundidades das memórias FIFO, as dimensões das RAMs e os endereços dos registos de monitorização e configuração). Como o modo de funcionamento do MICTP é definido pelo utilizador, os parâmetros de configuração estão agrupados e bem identificados. Por último, optimizou-se a síntese do MICT no que respeitava à área de implantação, e tirou-se partido das características específicas da FPGA mesmo sacrificando ligeiramente a portabilidade do código para dispositivos de outros fabricantes, sem perder de vista os atrasos de propagação que poderiam comprometer o bom funcionamento lógico.

4.1. Simulações *Funcional* e *Post-Layout*

O *firmware* foi simulado com sucesso na ferramenta *ModelSIM*, versão 6.4a. Para simular o envio de eventos foi desenvolvido em VHDL um modelo simples do *backplane* MIBAK e do módulo MIROD e utilizou-se um modelo em VHDL do MIOCT previamente desenvolvido[6]. O *firmware* destes modelos ficou disponível para que qualquer projectista que modifique o MICTP possa comparar os resultados.

Verificou-se que o código VHDL desenvolvido é compatível com o código dos controladores VME e I^2C existente no CERN, que as várias funções do MICTP estão correctas, satisfazendo as especificações do projecto, que o MICTP funciona adequadamente nos vários cenários de configuração possíveis e que a sua funcionalidade não é comprometida com a implementação física na FPGA.

Os resultados da simulação funcional corresponderam com aqueles obtidos com a simulação *post-layout*, onde se utilizou o ficheiro *SDF* gerado pela ferramenta *Quartus II* que contém a informação temporal sobre os atrasos dos sinais devidos à propagação ao longo das interligações e ao processamento pelos elementos eléctricos/lógicos do circuito.

[6] Quando o desenvolvimento do MICTP terminou, os outros módulos não estavam ainda terminados , por isso, tivemos de recorrer à emulação do respectivo funcionamento feita com a ajuda dos modelos referidos.

Table 1–1. Stratix II FPGA Family Features

Feature	EP2S15	EP2S30	EP2S60	EP2S90	EP2S130	EP2S180
ALMs	6,240	13,552	24,176	36,384	53,016	71,760
Adaptive look-up tables (ALUTs) *(1)*	12,480	27,104	48,352	72,768	106,032	143,520
Equivalent LEs *(2)*	15,600	33,880	60,440	90,960	132,540	179,400
M512 RAM blocks	104	202	329	488	699	930
M4K RAM blocks	78	144	255	408	609	768
M-RAM blocks	0	1	2	4	6	9
Total RAM bits	419,328	1,369,728	2,544,192	4,520,488	6,747,840	9,383,040
DSP blocks	12	16	36	48	63	96
18-bit × 18-bit multipliers *(3)*	48	64	144	192	252	384
Enhanced PLLs	2	2	4	4	4	4
Fast PLLs	4	4	8	8	8	8
Maximum user I/O pins	366	500	718	902	1,126	1,170

Notes to Table 1–1:
(1) One ALM contains two ALUTs. The ALUT is the cell used in the Quartus® II software for logic synthesis.
(2) This is the equivalent number of LEs in a Stratix device (four-input LUT-based architecture).
(3) These multipliers are implemented using the DSP blocks.

Tabela 2. Tabela com os recursos das FPGAs Stratix II da Altera, incluindo o modelo utilizado para alojar o MICTP. Os blocos modulares nesta FPGA são denominados LABs ("Logic Array Blocks"), consistindo cada um deles em oito ALMs ("Adaptive Logic Modules").[7]

4.2. Síntese do código

A síntese do código foi realizada utilizando o compilador *Synplify*, versão C-2009.03. O alvo de síntese é a FPGA *Altera Stratix II EP2S60* [7]. Os recursos disponíveis neste dispositivo, juntamente com outros modelos da mesma família, são mostrados na Tabela 2. Estas ferramentas, instaladas em máquinas situadas no CERN, foram utilizadas remotamente enquanto o trabalho se desenrolou em Lisboa.

Os relatórios de síntese indicam que as Memórias 4K da FPGA são utilizadas para implementar as RAMs de teste das palavras de *trigger* e de *multiplicidade*. As FIFOs do módulo *Rout*, por terem pouca profundidade, utilizam os recursos de memória 512. A PLL referida em 3.2.2 é implementada com as *Fast PLLs*, o que seria de esperar visto gerar apenas um relógio com atraso fixo. A tabela 3 apresenta sumariamente os recursos da FPGA utilizados pelo módulo.

	Disponíveis	Utilizados (%)
ALMs	24716	10
Memórias 4K	255	18
Memórias 512	329	3
Fast Plls	8	13

Tabela 3. Recursos utilizados pelo MICTP.

O módulo foi sintetizado em dois cenários: com e sem a opção de *Resource Sharing* (RS) da ferramenta *Synplify* activada. Comparando os resultados obtidos nos dois casos (tabela 4) tem-se a noção do compromisso área/velocidade decorrente de cada opção, que neste caso não é significativo pois o MICTP ocupa apenas uma parcela reduzida da FPGA. Recorda-se que todos os relógios funcionam a 40 MHz, pelo que há muita folga relativamente aos limites indicados pela síntese. A título informativo pode dizer-se que está previsto que no SLHC (o acelerador que será a evolução do LHC) aquela frequência suba para 80 MHz, um valor ainda nitidamente bem dentro dos limites indicados na tabela 4.

Relógio	Opção RS Não	Opção RS Sim
BCi	394,0	394,9
BCKD_0	430,9	430,9
BCKDI	352,1	322,0
BCDK_2	188,6	188,6

Tabela 4. Frequência máxima (em MHz) dos relógios no MICTP. A opção RS não teve influência significativa nos recursos gastos na FPGA pelo MIOCT.

4.3. Especificações Temporais

A partir dos ficheiros *VQM* ("Verilog Quartus Mapped") e TLC gerados pelo compilador *Synplify*, foi gerado o *bitstream* que configura a FPGA através da ferramenta *Quartus II*. Os resultados obtidos com esta última, apresentados nas tabelas 5 e 6, permitem verificar os *"Clock Setup Times"*, i.e. os intervalos de tempo em que os valores lógicos nas entradas devem estar estabilizados antes da chegada do flanco do relógio e de *"Clock to Output Delays"*, os atrasos máximos dos sinais até estarem disponíveis nas saídas. Para entradas que possam estar disponíveis depois do flanco descendente, o valor obtido tem sinal negativo.

A título de exemplo, mostra-se na figura 7 o diagrama temporal correspondente a uma das muitas simulações dos blocos do MICTP que foram efectuadas, neste caso do bloco deran_Ctrl.

A verificação dos requisitos temporais permitiu-nos configurar o módulo MICTP numa FPGA da placa MUCTPI e proceder aos testes em laboratório. Na

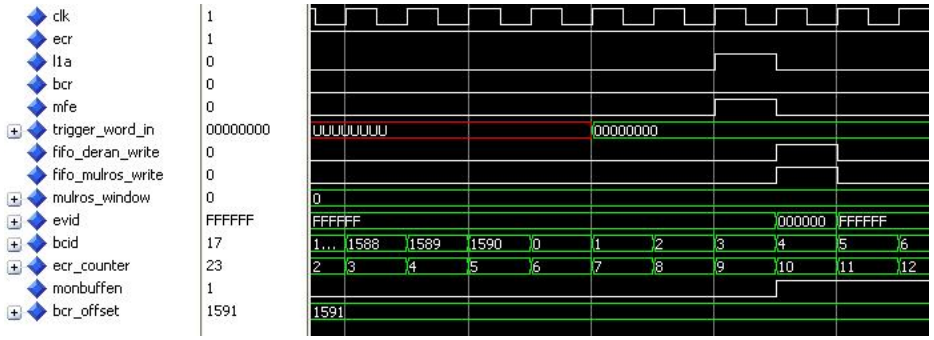


Figura 7. Exemplo de uma simulação, neste caso associada ao bloco deran_Ctrl.

Origem	CST	
e Entradas	restrições	obtidos
Delay25		
noxbc	Dv	7.050
MIBAK		
BCR	4.000	1.779
MBK_Busy_in	9.000	6.779
ECR	5.000	2.473
L1A	3.000	1.801
MFE	3.000	1.762
Palavra de		
Multiplicidade	3.000	1.779
token_MIROD	4.000	1.824
TST	4.000	1.776
Painel Frontal		
ECR	Dv	-1.530
L1A	Dv	-1.454
ORBIT	Dv	-1.545
TST	Dv	8.703
Palavra de		
trigger	4.0	1.783
VME		
Vme_addr	Dv	8.336
Vme_data_in	Dv	9.147
vme_read	Dv	8.553
Vme_write	Dv	7.001

Tabela 5. Temporização dos "Clock Setup Time" (ns). Dv corresponde a $T_{CK}/2$.

Módulo	COD	
e Saídas	restrições	obtidos
MICTP		
LEDs	Dv	4.098
Mul_Block		
mul_word_to_fp	Dv	7.802
Rout		
mbk_berr	8.000	5.288
mbk_dat	8.000	5.856
mbl_enabledriver	8.000	5.063
mbk_rdy_out	8.000	5.090
mbk_vld	8.000	4.977
token_MICTP	8.000	5.657
BUSY	Dv	6.459
Shap Block		
ECR	Dv	11.004
BCR	Dv	10.928
L1A	Dv	12.126
MFE	Dv	8.384
TST	Dv	8.703
Selxbc	Dv	6.581
VME		
Vme_data_out	Dv	6.893

Tabela 6. Temporização dos "Clock Output Delays" (ns).

Tese de Mestrado [8], encontra-se uma descrição bastante pormenorizada da síntese e da verificação do MICTP.

4.4. Validação em Laboratório

Na figura 8 vê-se a placa MUCTPI inserida num cesto VME do laboratório do CERN. Para além dos testes envolvendo apenas a funcionalidade do MICTP, foram testados alguns cenários, descritos mais adiante, para validar a comunicação com os módulos LTP (que gera os sinais temporais recebidos no painel do MUCTPI), CTP e MIOCT (através do MIBAK), e assim confirmar as funções lógicas executadas pelo MICTP. Desde já se adianta que em nenhum dos testes se detectou qualquer problema. É de notar que os constrangimentos deste processo e a dimensão e a complexidade dos blocos intervenientes (e.g. vários barramentos envolvidos no projecto, alguns módulos limítrofes ainda não implementados, impossibilidade de utilizar testadores convencionais e geração automática de vectores de teste), impediram o uso de outra metodologia estruturada de teste que não fosse a de exercitar o projecto no laboratório em todos os cenários previsíveis.

Validação do MICTP: a comunicação mais crítica dá-se entre o MICTP e o *delay25* através do barramento I^2C. Sendo este o circuito responsável pela geração dos sinais de relógio utilizados no MICTP e nos restantes componentes da PCB, se a comunicação falhar nenhum dos componentes da PCB funciona. Têm de ser habilitadas todas as linhas de relógio e configurados os atrasos. Outros aspectos validados foram a escrita nas RAMs de testes, a habilitação da Win_flag, a recepção do sinal de relógio *BCo* e a leitura dos registos do MICTP.

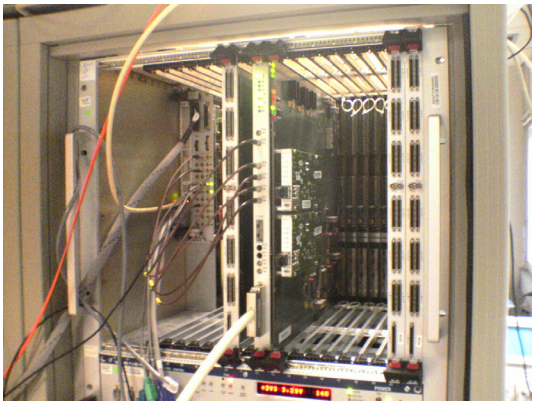


Figura 8. A placa MUCTPI num laboratório do CERN.

Formatação e Emissão dos Sinais: os sinais emitidos pelo LTP devem ser modificados pelo bloco Shap antes de seguirem para o MIBAK. O *backplane* MIBAK distribui-os para todo o sistema MUCTPI e reenvia-os para o MICTP. Como os sinais recebidos pelo MICTP são previamente definidos, podemos prever quais os valores nos contadores dos MIOCTs a partir da configuração dada a cada um dos blocos do módulo *Shap*. Foram testadas várias configurações para a formatação dos sinais, e em todos elas os contadores dos MIOCTs continha, o resultado esperado.

Recepção e Envio das Multiplicidades: as multiplicidades calculadas no MIOCT são recebidas pelo módulo Mul_Block que as remete para o CTP. As multiplicidades recebidas no CTP podem ser observadas com a função *snapshot* disponível no *software* usado nos testes. Comparando estas com as multiplicidades calculadas pelos MIOCTs e consultando os valores guardados nas tabelas, valida-se a transmissão realizada pelo MICTP.

Formação e Monitorização de Eventos: na altura da realização dos testes o módulo MIROD não estava ainda implementado na FPGA, e como tal não foi possível avaliar o envio de eventos formados pelo MICTP através do MIBAK. Contudo, configurou-se o MICTP para que todos os eventos formados fossem também guardados na FIFO Monitoring, sendo assim possível monitorizar os eventos e avaliar a informação neles enviada. O processo de formação e monitorização dos eventos envolve todos os módulos do MICTP, e por isso é necessário ter em atenção diversos detalhes, dos quais se destacam: a informação contida dos Eventos, o número de palavras por evento, as palavras de Multiplicidade e de *Trigger* usadas e a consulta dos registos de monitorização do MICTP.

Não foram detectados erros em quaisquer testes. Em última análise, eles validaram o correcto funcionamento das funções do MICTP, com a excepção do envio de eventos para o MIROD (pois este não estava disponível), satisfazendo os requisitos de projecto.

5. Conclusões

Este documento descreve trabalho desenvolvido ao longo de um ano e meio, que consistiu da migração do MICTP, um importante módulo digital inserido no MUCTPI, um sistema situado na interface da lógica dos detectores do ATLAS com o Central Trigger Processor. O trabalho exigiu um detalhado estudo dos vários módulos situados na fronteira do MICTP, dos protocolos dos barramentos VME e I^2C, do compilador *Simplify*, da ferramenta de desenvolvimento *Quartus II* da Altera e do software de configuração e de comunicação com os módulos do LVL1 realizado no CERN.

No final constatou-se que o código escrito em VHDL satisfez todos os requisitos funcionais e temporais preestabelecidos, visto todas as simulações *pre-* e *post-layout*, bem como todos os testes realizados em laboratório com o *firmware* do MICTP implementado na FPGA, terem sido bem sucedidos.

Agradecimentos

A realização deste trabalho beneficiou de uma bolsa enquadrada no Projecto CERN/FP/83551/2008 da FCT e de uma comparticipação directa do CERN para financiar parcialmente a estadia do principal autor nas suas instalações em Genebra, na Suíça.

Referências

[1] G. Schuler. *The MICTP Module of the Muon-CTP-Interface Demonstrator - Users Guide*, 2007. `https://twiki.cern.ch/twiki/bin/viewfile/Atlas/LevelOneCentralTriggerOperation?rev=1;filename=mictp.pdf?`

[2] ATLAS Collaboration. *ATLAS First-Level Trigger: Technical Design Report*, volume TDR-12 of *Technical Design Report ATLAS*. CERN, Geneva, 1998.

[3] Bruno Fernandes, Stefan Haas, Per Klöfver, Guiomar Evans, José Soares Augusto, and António Amorim. Desenvolvimento do Firmware do Módulo MICTP do Trigger de Nível 1 da Experiência ATLAS do LHC. *V Jornadas sobre Sistemas Reconfiguráveis - REC'2009*, 2009.

[4] H. Correia, A. Marchioro, P. Moreira, and J. Schrader. *Delay25, a 4 channel 1/2 ns programmable delay line*, 2005. CERN - EP/MIC, Geneva Switzerland.

[5] Richard Herveille. *I^2C - Master Core Specification*, 2003.

[6] Ralf Spiwoks. *Implementation of the VMEbus Interface of the ATLAS Level-1 Central Trigger Processor*, 2003. ATLAS Internal Note, ATL-DA-ES-0037, Version 1.1.

[7] *Stratix II Device Handbook, Vol. 1, V4.2*, 2007. `http://www.altera.com/literature/lit-stx2.jsp`

[8] Bruno Jesus Fernandes. *Projecto, Validação e Concretização do Módulo MICTP do Primeiro Nível do Filtro de Eventos do Detector ATLAS*. Dissertação de Mestrado em Eng. Física, Dep. de Física da Fac. de Ciências da Un. de Lisboa, 2009.

Scalable Accelerator Architecture for Local Alignment of DNA Sequences

Nuno Sebastião, Nuno Roma, Paulo Flores
INESC-ID / IST-TU Lisbon
Rua Alves Redol, 9, Lisboa
PORTUGAL
{*Nuno.Sebastiao, Nuno.Roma, Paulo.Flores*} *@inesc-id.pt*

Abstract

The Smith-Waterman algorithm is widely used to determine the optimal sequence alignment between two DNA sequences. This paper presents an innovative method to significantly reduce the computation time and memory space requirements of the traceback phase of this alignment algorithm. It also presents a flexible and scalable hardware architecture for accelerating such method, which can be easily expandable by the interconnection of several FPGAs. The results obtained from an implementation using a Virtex-5 FPGA showed that the proposed method is highly feasible in order to provide significant gains in terms of the overall performance of the whole alignment procedure when long sequences are processed. The obtained results also showed that it is preferable to span the array of processing elements through several FPGAs, rather than reusing the hardware resources of the individual array.

Keywords DNA; Local Sequence Alignment; Hardware Accelerator; FPGA

1. Introduction

With the recent advances in sequencing technologies, which allow the determination of the nucleotide sequence of the Deoxyribonucleic Acid (DNA), biologists gained access to an enormous amount of data. However, the DNA sequence size of most living cells can be quite large. For example, the size of the human DNA can be as large as 3×10^9 base pairs. This means that for each complete human individual genome that is sequenced, an additional dataset of 3×10^9 base pairs will be available for researchers. Such datasets are usually stored in databases to which biologists submit the newly sequenced DNA segments. One of these well known public databases is the GenBank [1]. The size of this database has doubled approximately every 18 months and the version released on June 15th, 2009, had approximately 105×10^9 base pairs.

The information contained in the DNA sequences is mainly extracted by homology, therefore requiring a large number of comparisons between sequences. However, exact search of a given sequence in the whole sequences database is often unfeasible due to the frequent mutations to which DNA is affected (nucleotide insertion, deletion and substitution). To overcome this complication, several techniques have been devised to find the optimal position where as many as possible nucleotides are found in the same positions. These methods, denoted by sequence alignment algorithms, are used to determine which sequences match more closely and how they align in order to show the zones that are common.

The alignments can be classified as either local or global. In global alignments, the complete sequences are aligned from one end to the other, whereas in local alignments only the subsequences that present the highest similarity are considered in the alignment. The local alignment is generally preferred when searching for similarities between distantly related biological sequences, since this type of alignment more closely focuses on the subsequences that were conserved during evolution.

The computational effort to perform such tasks in such a large dataset poses considerable challenges. The Dynamic Programming (DP) algorithm to find the optimal local sequence alignment between any two sequences has $O(nm)$ time complexity, where n and m denote the sizes of the sequences being aligned. Alternative sub-optimal heuristic algorithms, like BLAST, have been proposed to reduce the runtime. However, they may miss the optimal alignments between the sequences. Therefore, the use of the optimal alignment algorithms is usually preferred but not always performed due to the excessive runtime.

The use of hardware accelerators based on Field Programmable Gate Arrays (FPGAs) for High Performance Computing has been increasing over the past few years. Several algorithms have been accelerated with specialized architectures that were implemented in these devices. One of such algorithms is the Smith-Waterman (SW) algorithm [2], which uses DP to determine the optimal local alignment between any two sequences with $O(nm)$ complexity.

Several accelerator architectures have been proposed to implement the Smith-Waterman algorithm in FPGAs [3]. The most common architecture is based on a systolic array of Processing Elements (PEs). An example of a bidimensional systolic array, described using VHDL, is presented in [4]. Nevertheless, unidimensional (linear) systolic arrays are more commonly adopted [5, 6]. Some of these

accelerators can take advantage of the reconfiguration capabilities provided by FPGAs to optimize the PEs to the particular conditions of a given alignment [5]. Another implementation, which is available as a commercial solution, was developed by *CLC bio* [7]. The offered product also makes use of a FPGA to accelerate the matrix fill stage of the Smith-Waterman algorithm.

However, most of these accelerators have only focused on the part of the algorithm that calculates the alignment score. The alignment, itself, is usually obtained in a post-processing stage (usually implemented in a general purpose processor) where the scores are recalculated for the highest scoring sequences, by saving additional information that is required to retrieve the best alignment. In this paper, a new and more efficient method is proposed that makes use of the information obtained during the calculation of the alignment scores (in hardware), in order to reduce the time required to determine the alignment. To implement such technique, a scalable architecture that also enables the interconnection of several accelerators is presented and implemented in FPGA, thus allowing the use of a larger number of processing elements to permit a higher throughput.

This paper is organized as follows: In Section 2 it is presented the SW algorithm, which is used to determine the optimal local alignment. Section 3 presents the architecture used to accelerate the local alignment procedure. Section 4 shows the obtained results in an FPGA. The conclusions are presented in Section 5.

2. Local alignment

Considering two strings S_1 and S_2 of an alphabet Σ with sizes n and m, respectively, a local alignment reveals which pair of substrings of sequences S_1 and S_2 optimally align, such that no other pairs of substrings have a higher similarity score. A commonly used algorithm to determine the local alignment is the SW algorithm, which has a $O(nm)$ time complexity [2]. This algorithm uses a DP method composed of three essential parts: the recurrence relation, the matrix computation and the traceback [8].

2.1. Smith-Waterman Algorithm

Let $G(i,j)$ represent the best alignment score between a suffix of string $S_1[1..i]$ and a suffix of string $S_2[1..j]$. The SW algorithm allows the computation of $G(n,m)$ (the local alignment between the two strings) by recursively calculating $G(i,j)$ (the local alignment between prefixes of S_1 and S_2).

The recursive relations to calculate the local alignment score $G(i,j)$ are given by Equation 1

$$G(i,j) = \max \begin{cases} G(i-1,j-1) + Sbc(S_1(i),S_2(j)), \\ G(i-1,j) - \alpha, \\ G(i,j-1) - \alpha, \\ 0 \end{cases} \quad (1)$$

The $Sbc(S_1(i),S_2(j))$ function denotes the value obtained by aligning character $S_1(i)$ against character $S_2(j)$. This value represents the substitution score. The α value represents the gap penalty cost (the cost of aligning a character to a space). An example of a substitution function is shown in Table 1.

Table 1: Example of a substitution score matrix.

Sbc	A	C	G	T
A	3	-1	-1	-1
C	-1	3	-1	-1
G	-1	-1	3	-1
T	-1	-1	-1	3

The alignment scores are usually positive for characters that match, thus denoting a similarity between the two. On the contrary, mismatching characters may have either positive and negative scores, according to the type of alignment that is being performed, denoting the biological proximity between the two. In contrast, the gap penalty cost α is always positive. Different substitution score matrices are used to reveal different alignments. The particular score values defined in these matrices are determined by biologists according to evolutionary relations.

The initial conditions for the calculation are the following:

$$G(i,0) = G(0,j) = 0$$

After filling the entire matrix G, the substrings of S_1 and S_2 that best align are found by first locating the cell with the highest score in G. Then, all matrix cells that lead to this highest score cell are sequentially determined by performing a traceback procedure. The traceback procedure ends when a cell with a score of zero is reached. Such traceback identifies the substrings as well as the corresponding alignment. The path taken at each cell is chosen based on which of the three neighboring cells (left, top-left and top) was used to calculate the current cell value based on the recurrence equations (eq. 1). When the neighbor is the left cell ($G(i,j-1)$) then this corresponds to inserting a space (opening a gap) in S_1 at position i. If it was the top cell ($G(i-1,j)$), then this corresponds to inserting a space (opening a gap) in S_2 at position j. When the neighbor is the top-left cell ($G(i-1,j-1)$) then this corresponds either to a match or to a substitution. The traceback phase has a $O(n+m)$ time complexity.

Table 2 shows an example of the calculated score matrix for aligning two sequences ($S_1 = CAGCCTCGCT$ and $S_2 = AATGCCATTGAC$) using the substitution score matrix presented in Table 1, where a match has a score of 3 and a mismatch a score of -1. A gap has a penalty of 4. The shadowed cells in the table represent the traceback path that was taken to determine the best alignment, which is shown in Figure 1.

978-972-789-304-1 REC'2010

Table 2: Example of an alignment score matrix.

		0	1	2	3	4	5	6	7	8	9	10	11	12
	G	ø	A	A	T	G	C	C	A	T	T	G	A	C
0	ø	0	0	0	0	0	0	0	0	0	0	0	0	0
1	C	0	0	0	0	0	3	3	0	0	0	0	0	3
2	A	0	3	3	0	0	0	2	6	2	0	0	3	0
3	G	0	0	2	2	3	0	0	2	5	1	3	0	2
4	C	0	0	0	1	1	6	3	0	1	4	0	2	3
5	C	0	0	0	0	0	4	9	5	1	0	3	0	5
6	T	0	0	0	3	0	0	5	8	8	4	0	2	1
7	C	0	0	0	0	2	3	3	4	7	7	3	0	5
8	G	0	0	0	0	3	1	2	2	3	6	10	6	2
9	C	0	0	0	0	0	6	4	1	1	2	6	9	9
10	T	0	0	0	3	0	2	5	3	4	4	2	5	8

$$\begin{array}{cccccccc} G & C & C & A & T & T & G \\ | & | & | & & | & & | \\ G & C & C & \_ & T & C & G \end{array}$$

Figure 1: Obtained alignment.

2.2. Tracking of the Origin and End alignment indexes

When only the alignment score is required, it is not necessary to perform the traceback phase of the SW algorithm. However, whenever the alignment between the sequences must also be determined, the traceback phase must be implemented. However, most hardware accelerators that have been proposed for the alignment algorithms only implement the matrix computation (without performing the traceback phase). Therefore, only the alignment score is calculated by the accelerator. Afterwards, whenever the alignment score is greater than a given threshold, the whole G matrix is recalculated (usually by using a general purpose processor) maintaining enough intermediate data to perform the traceback and retrieve the corresponding alignment. Hence, the recalculation of the entire G matrix is performed outside the accelerator without keeping any data from the previously calculated matrix score.

However, it can be shown that the time and memory space that is required to find the local alignment can be significantly reduced. In fact, and considering a given pair of sequences S_1 and S_2, if it is possible to know that the local alignment starts in characters at position $S_1(p)$ and $S_2(q)$ represented as (p,q) and ends in characters at position $S_1(u)$ and $S_1(v)$ represented as (u,v), then the local alignment can be obtained by just recalculating the alignment between the subsequences $S_a = S_1[p..u]$ and $S_b = S_2[q..v]$.

As an example, from the data shown in Table 2, it is possible to determine that the alignment starts in characters at position $(3,4)$ and ends in the characters at position $(8,10)$. With this information, the optimal local alignment between S_1 and S_2 can be found by only calculating the alignment between subsequences $S_a = S_1[3..8] = GCCTCG$ and $S_b = S_2[4..10] = GCCATTG$. The alignment between S_a and S_b can now be determined by computing a much smaller G matrix and performing the traceback, as shown in Table 3.

Table 3: Reduced alignment score matrix.

G	ø	G	C	C	A	T	T	G
ø	0	0	0	0	0	0	0	0
G	0	3	0	0	0	0	0	3
C	0	0	6	3	0	0	0	0
C	0	0	3	9	5	1	0	0
T	0	0	0	5	8	8	4	0
C	0	0	3	3	4	7	7	3
G	0	3	0	2	2	3	6	10

Hence, the advantage of this method resides in the fact that the time and memory space required to recompute the G matrix for the subsequences that participate in the alignment is usually significantly reduced when compared to the entire sequences. Consequently, this method also reduces the computational effort of the alignment algorithm.

To determine the character positions where the alignment starts an auxiliary matrix, C_b, will be used. Let $C_b(i,j)$ represent the coordinates of the matrix cell where the alignment of string $S_1[1..i]$ and string $S_2[1..j]$ starts. Using the DP method that is used to calculate $G(i,j)$, it is possible to simultaneously build a matrix C_b, with the same size as G, that maintains a track of which cell originated the score that reached cell (i,j).

The recursive relations that determine the coordinates of the matrix cell that originated the alignment ending at cell (i,j) are given by Equation 2.

The initial conditions for the calculation are:

$$C_b(i,0) = C_b(0,j) = (0,0)$$

With this method, it is possible to find, at cell $C_b(i,j)$, the coordinates of the cell where the alignment ending at cell $G(i,j)$ was originated. Afterwards, by knowing the cell where the maximum score occurred, $G(u,v)$, it is possible to determine from $C_b(u,v) = (p,q)$ the coordinates of the cell where the alignment began. Then, to obtain the desired alignment, the score matrix has to be rebuilt only for the subsequences $S_1[p..u]$ and $S_2[q..v]$ which are usually considerably smaller than the entire S_1 and S_2 sequences.

An example of table C_b for the alignment of sequences S_1 and S_2, whose G matrix was presented in Table 2, is shown in Table 4. In this example, by knowing from the G matrix that the maximum score occurs at cell $(8,10)$, it is possible to retrieve the coordinates of the beginning of the alignment in cell $C_b(8,10) = (3,4)$.

3. Architecture

The local alignment algorithm described in Section 2 is usually applied to biological sequences in which $m \gg n$ (e.g. $n \approx 500$ and $m \approx 10^6$). The matrix fill stage of this algorithm is the most computationally intensive and is therefore a good candidate for parallelization. However, the data dependencies that exist to calculate each matrix cell value

$$C_b(i,j) = \begin{cases} (i,j), & if\ G(i,j) = G(i-1,j-1) + Sbc(S_1(i),S_2(j))\ and\ C_b(i-1,j-1) = (0,0) \\ C_b(i-1,j-1), & if\ G(i,j) = G(i-1,j-1) + Sbc(S_1(i),S_2(j))\ and\ C_b(i-1,j-1) \neq (0,0) \\ C_b(i-1,j), & if\ G(i,j) = G(i-1,j) - \alpha, \\ C_b(i,j-1), & if\ G(i,j) = G(i,j-1) - \alpha, \\ (0,0), & if\ G(i,j) = 0 \end{cases} \quad (2)$$

Table 4: Example of an Origin and End Alignment Indexes tracking matrix.

		0	1	2	3	4	5	6	7	8	9	10	11	12
	C_b	ø	A	A	T	G	C	C	A	T	T	G	A	C
0	ø	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)
1	C	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(1,5)	(1,6)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(1,12)
2	A	(0,0)	(2,1)	(2,2)	(0,0)	(0,0)	(0,0)	(1,5)	(1,6)	(1,6)	(0,0)	(0,0)	(2,11)	(0,0)
3	G	(0,0)	(0,0)	(2,1)	(2,2)	(3,4)	(0,0)	(0,0)	(1,6)	(1,6)	(1,6)	(3,10)	(0,0)	(2,11)
4	C	(0,0)	(0,0)	(0,0)	(2,1)	(2,2)	(3,4)	(4,6)	(0,0)	(1,6)	(1,6)	(0,0)	(3,10)	(4,12)
5	C	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(2,2)	(3,4)	(3,4)	(3,4)	(0,0)	(1,6)	(0,0)	(3,10)
6	T	(0,0)	(0,0)	(0,0)	(6,3)	(0,0)	(0,0)	(3,4)	(3,4)	(3,4)	(3,4)	(0,0)	(1,6)	(3,10)
7	C	(0,0)	(0,0)	(0,0)	(0,0)	(6,3)	(7,5)	(7,6)	(3,4)	(3,4)	(3,4)	(3,4)	(0,0)	(1,6)
8	G	(0,0)	(0,0)	(0,0)	(0,0)	(8,4)	(6,3)	(7,5)	(7,6)	(3,4)	(3,4)	(3,4)	(3,4)	(3,4)
9	C	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(8,4)	(6,3)	(7,5)	(7,6)	(3,4)	(3,4)	(3,4)	(3,4)
10	T	(0,0)	(0,0)	(0,0)	(10,3)	(0,0)	(8,4)	(8,4)	(6,3)	(7,5)	(7,6)	(3,4)	(3,4)	(3,4)

(to calculate the value for cell $G(i,j)$ it is necessary to know the values of $G(i-1,j-1)$, $G(i,j-1)$ and $G(i-1,j)$) highly restrict the parallelization to the simultaneous computation of the values along the matrix anti-diagonal direction.

Specialized parallel hardware that is capable of performing a high number of simultaneous matrix computations is especially suited for this task. A linear systolic array with several identical PEs, as shown in Figure 2, is an efficient architecture to implement this type of computation, by simultaneously computing the values of the *G* matrix that are located in a given anti-diagonal.

3.1. Processing Element

The PEs architecture described in this paper is based on the PEs architecture presented in [5]. The simplest PE only implements the function of the basic local alignment algorithm and is shown in Figure 3. It has a two stage pipelined datapath to calculate a score matrix cell value (output in $G(i,j)$). The throughput of each element is one score value per clock cycle. Since the Smith-Waterman algorithm requires the determination of the maximum score value throughout the entire matrix, it is necessary to have an additional datapath that selects the maximum score that has been calculated in the array (output $Max(i,j)$). The PE *i* selects and stores the maximum score that was computed by PEs 1 through *i*.

The array evolves along the line by shifting the reference sequence character symbols through the PEs. In this array, the character $S_1(i)$ is allocated to the *i*th PE and this PE performs, at every clock cycle, the computations required to determine the score value of a certain matrix cell. This computation involves, among other operations, determining the substitution score between two characters (the value

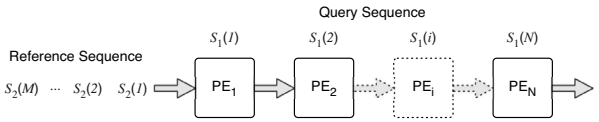


Figure 2: Systolic array structure.

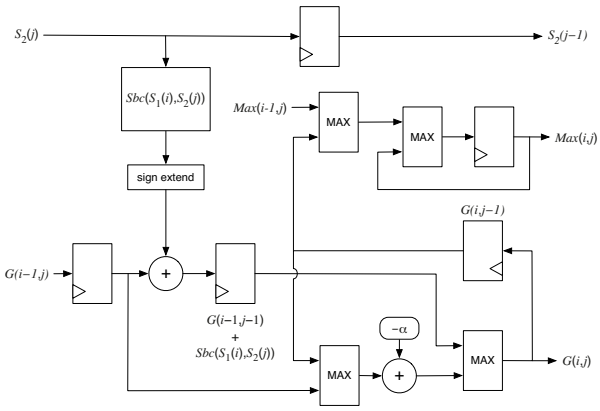


Figure 3: Simple PE architecture.

978-972-789-304-1 REC'2010

of $Sbc(S_1(i), S_2(j))$. Since each PE performs the operations only over one character of S_1, it only needs to store the column of the substitution cost matrix that represents the costs of aligning character $S_1(i)$ to the entire alphabet. The computation of the matrix cell value $G(i, j)$ also requires the determination of the maximum values that are the result of the three distinct possibilities presented in Equation 1. The zero condition of the Smith-Waterman algorithm is implemented by controlling the reset signal of the registers that store the value $G(i, j)$, by using the most significant bit (sign bit) of the score value, i.e., if the maximum value among the three partial scores is negative, then it clears the registers that hold that given score value.

After all the reference sequence (S_2) characters have passed through all the PEs, the alignment score is available at $Max(i, j)$ output of the last PE.

3.2. Array programming

The query sequence (S_1) data which is loaded into the array is the substitution score matrix column that corresponds to the symbol at that position. In fact, since each PE only performs comparisons to a given query sequence character, it will just access the values present in a certain matrix column. Therefore, each PE will only receive the substitution score matrix column that corresponds to the query sequence character allocated to that PE.

Within each PE, such data is stored using dedicated registers since this allows for a fast reprogramming of the PEs for a new query sequence. In the event of a PE is not being used (because the query sequence has a smaller size than the number of PEs (N)), the substitution score data that is stored in such PE corresponds to a substitution matrix column in which every value is zero.

To program the query sequence (S_1) score values, an auxiliary structure was included in the array. This structure is composed by a n bit-width shift register that allows to shift the values of a substitution matrix column through the several PEs. This approach provides the load operation of a *new* query sequence into this temporary storage shift register, by serially shifting the substitution matrix column data while the array is processing the data regarding the *current* query sequence. As soon as the array has finished processing the data regarding the *current* query sequence, the *new* query sequence data, which is stored in the auxiliary shift register, is parallel loaded (in just one clock cycle) into the respective PEs. This allows to mask the time that would be required to shift the *new* query sequence data into the array, while the array is processing the *current* data and therefore, it ends-up by programming the actual query sequence in just one clock cycle, which significantly reduces the amount of time required for programming the array.

To allow the usage of the same array to process query sequences (S_1) larger than the number of available PEs (N), it is possible to store intermediate results in a local memory. These results are the output values of the last PE in the array and correspond to the scores of a complete row of matrix G. The size of this memory limits the size of the reference sequence (S_2), since it must entirely fit, along with the intermediate calculation data, in this memory. This memory is organized as a FIFO memory and the values stored on it will be later reintroduced in the array and used to compute the alignment for larger sequences.

3.3. Tracking of the Origin and End Alignment Indexes

As it was previously referred, typical applications of hardware accelerators for sequence alignment focus on accelerating only the matrix computation, leaving the traceback for a posterior phase. Furthermore, such implementations only return the alignment score between the two sequences and not all the values of matrix G. Therefore, to obtain the actual alignment, these accelerators force the recomputation of the entire matrix (using a general purpose processor) to be able to perform the traceback phase. This recomputation (and subsequent traceback) is performed in the cases when the alignment score, calculated by the accelerator, is above a given threshold which is defined by the user.

The proposed architecture avoids the recomputation of the entire G matrix by propagating through the PEs, not only the partial maximum scores in the matrix, but also the coordinates where such scores had their origin (the beginning of the alignment), together with the coordinates where the maximum score occurred. As it was shown in Section 2.2, this enables the recomputation phase of matrix G to only focus on the substrings that are actually involved in the alignment and avoid the recomputation of the whole matrix G. Thus, the time and memory space requirements to obtain the sequence alignment are substantially reduced.

To achieve this, an enhanced PE, whose architecture is presented in Figure 4, was developed with the hardware necessary to implement the calculation and propagation of matrix C_b. The datapath that implements this computation is similar to the datapath of a simple PE. The decision logic (inside the maximum calculation units - *Max*) is also used, in this case, to control the selection units of the Origin and End Alignment Indexes (OEAI) tracking coordinates. In each PE, the origin index coordinates, which indicate where the alignment began, are propagated based on the conditions shown in Equation 2.

Since only the coordinates of the origin cell need to be selected alongside with the scores, the PE only incorporates hardware resources to implement such selection in the score calculation datapath. Furthermore, since the simple PEs array is not capable of determining and keep track of the location of the maximum score cell, additional hardware was also included, in the maximum selection datapath, to support the propagation of the coordinates of the cell where the maximum value occurred. Within each PE, the coordinates of the current cell are obtained by using the PE index (i) and a symbol coordinate (j) that comes alongside with the symbol that is at the input of PE_i.

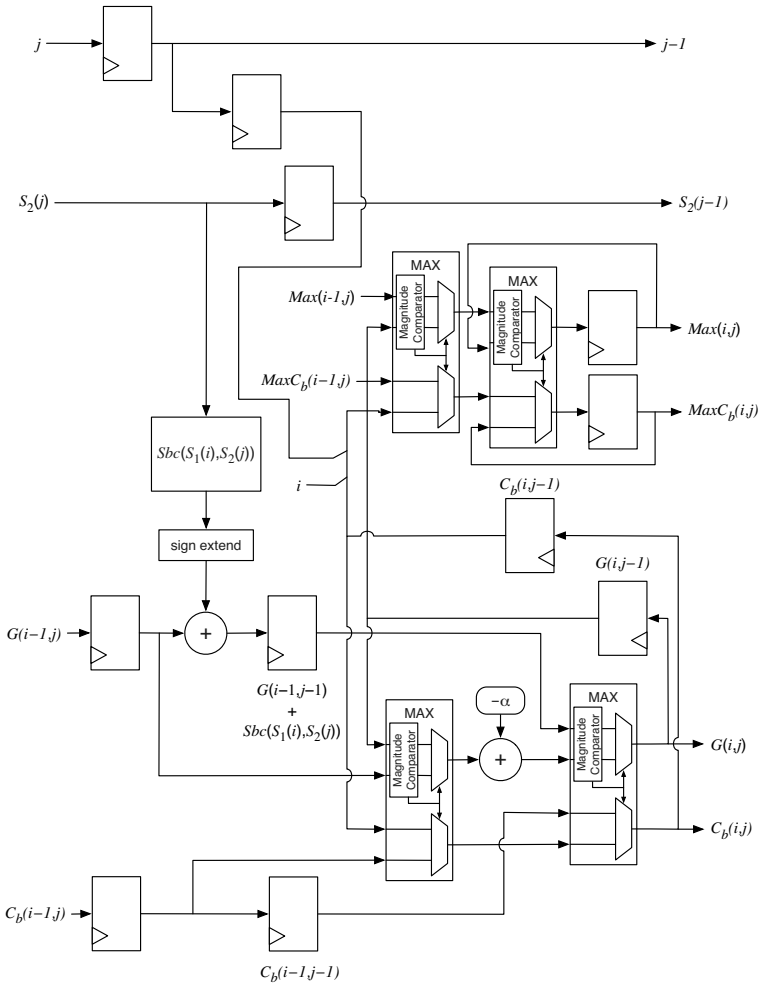


Figure 4: Architecture of PE with OEAI tracking.

3.4. Scalability and Reconfigurability

Whenever the query sequences (S_1) to be aligned are larger than the number of PEs that fit in a FPGA, it is necessary to either reuse or expand the array. Both of these capabilities are supported by the proposed architecture.

When the array is reused in order to perform the alignment with query sequences longer than the number of available PEs, an additional set of control hardware and memory are included in the architecture. The added memory is used to store all the information of a single row of the G matrix (and of the C_b matrix, in case the OEAI tracking function is used). This enables to compute an entire horizontal section of the G matrix, which corresponds to aligning a segment of the query sequence with the entire reference sequence. Afterwards, a new segment of the query sequence is loaded into the PEs and the next horizontal section of the G matrix is computed. This process is repeated until the query sequence has ended. With this implementation, the array limits the size of the reference sequence (m), since the complete data of a single row of matrix G ($m+1$ elements) must fit in a memory block that is available in the device (FPGA). Since the available memory blocks inside this type of devices are usually not large, this capability is only advised for alignments in which the reference sequence is not too long.

To cope with simultaneous long query and reference sequences, this architecture also allows to span the array of PEs over more than one FPGA. This allows to increase the number of PEs in the array, therefore providing the computation of alignments with longer query sequences and without constraining the size of the reference sequence to the amount of available memory inside the FPGA. To implement this capability, relatively small FIFO memories, which store the outputs of the last PE, are used as buffers for the communication between the FPGAs and high-speed communication links are also used to enable a high-speed connection between the devices (see Figure 5).

Moreover, by taking advantage of the reconfiguration capabilities of the FPGA, it is possible to generate systolic array structures that have the number of PEs adapted to the size of the query and reference sequences that will be aligned. Therefore, the reconfiguration capability of the FPGA allows to maximize the obtained performance for a given set of query sequences.

Table 5: Obtained results when using a single FPGA with simple PEs.

# PEs	Symbol bit-width	Score bit-width	Occupied Slice Registers	Occupied Slice LUTs	Maximum Frequency [MHz]	Maximum Throughput [GCUPS]
16	2	8	928 (0.4%)	1656 (0.8%)	205	3.2
16	2	16	1456 (0.7%)	2997 (1.4%)	173	2.7
128	2	11	9088 (4.4%)	19653 (9.5%)	201	25.7
256	2	12	19200 (9.3%)	41919 (20.2%)	171	43.8
512	2	13	40448 (19.5%)	88246 (42.6%)	155	79.0

Table 6: Obtained results when using Origin and End Alignment Indexes tracking.

# PEs	Score bit-width	Maximum Reference size	Maximum Query size	Occupied Slice Registers	Occupied Slice LUTs	Maximum Frequency [MHz]	Maximum Throughput [GCUPS]
16	8	1024 (2^{10})	16	2309 (1.1%)	2816 (1.4%)	216	3.45
16	16	8192 (2^{13})	16	3156 (1.5%)	5187 (2.5%)	149	2.38
128	11	8192 (2^{13})	16	23732 (11.4%)	39822 (19.2%)	161	20.62
128	11	131×10^3 (2^{17})	128	28681 (13.8%)	46635 (22.5%)	162	20.72
256	12	134×10^6 (2^{27})	256	76546 (36.9%)	100489 (48.5%)	147	37.55

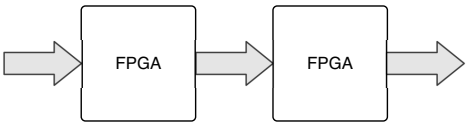


Figure 5: Array extension.

4. FPGA Results

The previously presented architecture was described using parameterizable VHDL code and synthesized for a Xilinx Virtex-5 FPGA (xc5vlx330t) using Xilinx ISE 9.2.04i. Initially, only the simple PE architecture was used, in order to evaluate the resource usage and performance of such array. The obtained results are shown in Table 5

The symbol bit-width represents the number of bits of the registers that hold the characters to be aligned. Since the results were obtained with DNA sequences, which are composed of only four different nucleotides, the characters can be encoded using only 2 bits. The score bit-width represents the number of bits of the registers that hold the score values of G. This resolution is determined according to the specific needs of the system and should have a value that guarantees that no overflows will occur during the processing of matrix G.

As it would be expected, the results show that the throughput increases with the number of PEs, despite the fact that the maximum operating frequency decreases when the device occupancy increases. The maximum obtained throughput is 79×10^9 CUPS (Cell Updates per Second) for a configuration with 512 PEs.

4.1. Tracking of the Origin and End Alignment Indexes

When using the OEAI tracking functionality, the hardware resources spent on each PE are increased, as it can be seen in Table 6. Therefore, for the same FPGA device, the maximum number of PEs that may be implemented in the device is reduced by a factor of 2, which may affect the throughput when large number of PEs are needed. On the other hand, when comparing arrays with the same number of PEs, the array that uses the PEs with the OEAI tracking functionality has a decrease in performance due to a slight reduction of the maximum operating frequency (e.g 14% decrease in maximum throughput for the array with 256 elements). Even so, in application environments where the FPGA resources are not a constraint, this decrease in peak performance may be largely compensated by the fact that the traceback phase of the Smith-Waterman algorithm will take significantly less time and the memory space requirements will be significantly reduced.

As an example, when aligning a query sequence with 200 characters against a reference sequence with 100×10^3 characters, matrix G will have a dimension of about 20×10^6 cells. By considering the scores given in Section 2.1 (a match has a score of 3 and a gap a score of -4), the maximum alignment size (including gaps) will be approximately 350 characters long (a maximum of 3 gaps can be inserted between a 4 character set of S_1 or S_2, thus expanding the size of the alignment by a maximum of $3/4$). Therefore, with the OEAI tracking functionality the maximum size of the alignment matrix G that needs to be recomputed during the traceback stage to find the best alignment has a size of $[(200 + (3/4) * 200/2)]^2 = 76 \times 10^3$ cells, when the gaps are evenly distributed among the two sequences. This leads to a $1/264$ reduction in the size of the recalculated

Table 7: FIFO sizes required for array reuse using OEAI tracking.

Score bit-width	Maximum Reference size	Maximum Query size	Total FIFO size (Bytes)
8	1024	16	$9,5 \times 10^3$
16	8192	16	104×10^3
11	8192	16	94×10^3
11	131×10^3	128	2.0×10^6
12	134×10^6	256	2.8×10^9

matrix leading to a quite significant decrease of the processing time and the involved memory requirements, which largely compensates the individual PE performance degradation described above.

When using the OEAI functionality, the Maximum Reference Sequence Size is imposed by the bit-width of the registers that hold the coordinates for the reference sequence. In contrast, the Maximum Query Size is imposed by the maximum number of PEs that can be accommodated.

4.2. Array reuse

When the reuse of the PEs array is considered to compute alignments with query sequences larger than the number of PEs, the main concern relates to the amount of memory required to store all the partial values of an entire row of the *G* matrix. Table 7 depicts the amount of memory that is required for several configurations.

As it can be seen, the memory requirements for aligning a query sequence against a reference sequence that has 131×10^3 characters requires about 2MB of memory to hold the values of a single row of *G*. For even larger reference sequences, the required amount of memory is so large that it will not be possible to reuse the array to perform alignments with query sequences larger than the number of PEs.

In such situations, it is preferable to use the expansion method (connecting another FPGA as shown in Figure 5) to enable fast and unconstrained alignments with large query sequences.

5. Conclusions

This paper presented a flexible architecture for accelerating the SW local sequence alignment algorithm using FPGAs. It also proposed an innovative method that provides a significant reduction of the computation time and memory space requirements of the traceback phase of the alignment procedure. The results obtained from an implementation of the proposed architecture using a Virtex-5 FPGA showed that such method is highly feasible in order to provide significant gains in terms of the overall performance of the whole alignment procedure. Furthermore, with long reference sequences and when the query sequences are longer than the number of PEs that can be accommodated in a device, it was shown that it is preferable to span the array of PEs across multiple FPGA devices instead of reusing the array. This is mainly a constraint imposed due to the limited amount of memory space available in current FPGAs.

Acknowledgment

This work has been partially supported by the PhD grant with reference SFRH/BD/43497/2008 provided by the Portuguese Foundation for Science and Technology.

References

[1] D. Benson, I. Karsch-Mizrachi, D. Lipman, J. Ostell, and E. Sayers, "GenBank," *Nucleic Acids Res.*, vol. 37, no. Database issue, pp. D26–D31, Jan. 2009.

[2] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Mol. Biol.*, vol. 147, no. 1, pp. 195–197, 1981.

[3] T. Ramdas and G. Egan, "A Survey of FPGAs for Acceleration of High Performance Computing and their Application to Computational Molecular Biology," in *TENCON 2005*, Nov. 2005, pp. 1–6.

[4] L. Hasan, Z. Al-Ars, Z. Nawaz, and K. Bertels, "Hardware implementation of the Smith-Waterman Algorithm using Recursive Variable Expansion," in *3rd Int. Design and Test Workshop, IDT 2008*, Dec. 2008, pp. 135–140.

[5] T. Oliver, B. Schmidt, and D. Maskell, "Hyper customized processors for bio-sequence database scanning on FPGAs," in *Proc. 13th Int. Symp. Field-programmable gate arrays, FPGA'05*. ACM, 2005, pp. 229–237.

[6] K. Benkrid, Y. Liu, and A. Benkrid, "A Highly Parameterized and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 4, pp. 561–570, Apr. 2009.

[7] "White paper on CLC Bioinformatics Cube 1.03," CLC Bio, Finlandsgade 10-12 - 8200 Aarhus N - Denmark, Tech. Rep., May 2007.

[8] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: computer science and computational biology*. Cambridge University Press, 1997.

Simulação em FPGA de Redes Reguladoras com Topologia Livre de Escala

Júlio C. G. Vendramini, Ricardo Ferreira, Leonardo Carvalho
Dpto Informática, Universidade Federal de Viçosa, Viçosa, 36570-000, Brazil
ricardo@ufv.br

Abstract

Este trabalho apresenta a implementação de um algoritmo para cálculo do período em redes reguladores de genes. As redes são modeladas por grafos booleanos livre de escala, e implementadas em um FPGA. A implementação é baseada em uma arquitetura genérica que permite que vários grafos sejam simulados dinamicamente sem a necessidade de resintetizar. Os vértices no FPGA incorporam uma máquina de estados que executa a simulação de cada vértice em paralelo. A comunicação é realizada através de uma rede multiestágio. O ganho em aceleração da implementação paralela em FPGA foi de 3 ordens de grandeza em relação a versão em software.

1. Introdução

Recentemente, vários trabalhos vem sendo propostos para analizar a dinâmica de sistemas biológicos através do estudo das redes regulatórias de genes [1-6]. Estes trabalhos são motivados pela disponibilidade de dados experimentais sobre a interação entre genes e proteínas. As redes regulatórias são modeladas por um grafo booleano, onde cada vértice v_i representa um gene, e pode ter dois estados ($e_i = 0$ ou 1). Uma aresta $a_{i->j}$ indica que o vértice v_i atua sobre o vértice v_j. Para cada vértice, uma função booleana calcula seu estado em função dos estados dos vértices incidentes. O estado do grafo é representado pelo conjunto de estados dos vértices $e_g=(e_0,..,e_i,..,e_n)$. A dinâmica do sistema estuda a evolução dos estados do grafo e a presença de ciclos. Um ciclo representa a estabilidade em uma rede regulatória. Modelos analíticos e simulações são usadas no cálculo dos ciclos [1-6].

Entretanto, uma rede com N vértices terá 2^n estados, que inviabiliza a exploração completa do espaço de soluções. Enquanto os modelos analíticos propõe simplificações [1], os modelos por simulação ficam limitados a pequenos valores de N [1,4,5,6]. Para cada passo de simulação é necessário visitar todas as arestas e vértices, ou seja, cada passo tem complexidade O(A+N). Neste trabalho apresentamos uma implementação em FPGA onde cada passo tem complexidade O(1), pois os vértices são atualizados em paralelo. Resultados experimentais mostraram uma aceleração de 600 à 1000 vezes da versão em FPGA em relação a versão em software para redes com 100 à 200 vértices.

Na seção 2 apresentamos as redes booleanas livre de escala. Na seção 3, um algoritmo para cálculo do ciclo é apresentado. A implementação em FPGA é detalhada na seção 4. Finalmente, os resultados e conclusões são apresentados na seção 5.

2. Redes Livre de Escala

Inicialmente, os modelos de rede booleana eram baseados em redes aleatórias e cada vértice tinha o mesmo número de vizinhos, denominado por modelo de Kauffman [1]. Posteriormente, os modelos livre de escala foram introduzidos [3,4,6], onde alguns vértices, denominados por hubs, concentram um maior número de arestas. Os grafos livre de escala vem sendo usados para redes sociais, internet, redes regulatórias [3,6]. A Fig. 1 apresenta duas instâncias de um grafo livre de escala. A cor escura representa o estado 1 e a cor branca o estado 0. O vértice v4 é um exemplo de hub.

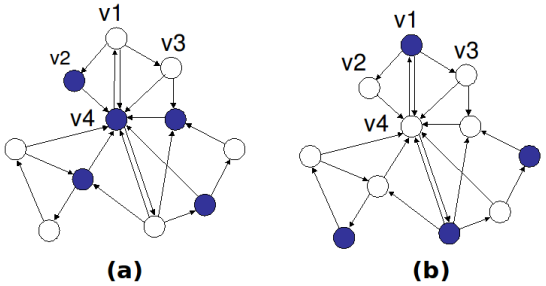


Figura 1. Dois estados de uma rede livre de escala

A Fig. 1 ilustra também a transição de estados do grafo, que passa do estado da Fig 1(a) para o estado da Fig. 1(b). Neste trabalho iremos considerar o

modelo síncrono [1,4,5,6], onde todos os vértices atualizam seu estado ao mesmo tempo.

3. Cálculo do Período

O comprimento do período ou do atractor da rede depende da topologia da rede, do estado inicial e da função de transição de cada vértice. Iremos supor um grafo livre de escala, que se aproxima dos modelos biológicos [3]. A função de transição será a função majoritária, que foi pouco explorada na literatura por gerar períodos maiores. Vamos supor a mesma função para todos os vértices. Analogamente aos outros trabalhos, a distribuição das arestas respeita a topologia livre de escala [4-6].

Considere o grafo da Fig. 1(a) no tempo t e Fig. 1(b) no tempo t+1. O vértice v1 está conectado ao vértice v4. A função majoritária, irá considerar o valor dominante (0 ou 1) no tempo t para ser o novo estado do vértice no tempo t+1, se a maioria for 0 (1), o estado será 0 (1). O estado do vértice v1 será 1, pois o vizinho v4 tem valor 1 no tempo t. Ao mesmo tempo o vértice v4 passa para estado 0, pois a maioria dos vizinhos tem o valor 0 no tempo t.

O algoritmo básico para cálculo do período [5] é baseado no determinismo do modelo síncrono, onde dado um estado da rede, o próximo estado é único. Duas simulações são realizadas S_0 e S_1. Uma simulação evolui com velocidade 1, $S_0(t+1) \leftarrow S_0(t)$, enquanto uma segunda simulação evolui com a velocidade dobrada $S_1(t+2) \leftarrow S_1(t+1) \leftarrow S_1(t)$. Quando os estado em $S_0(t_0)$ e $S_1(t_1)$ se tornam idênticos, a simulação para. S_1 fica estacionado em t_1, enquanto a simulação em S_0 evolui até $S_0(t_0+P)=S_1(t_1)$. O número de passos para S_0 e S_1 se encontrarem novamente, determina o período P.

Suponha que o grafo esteja em um estado inicial A e que durante a simulação passe pela seguinte sequência A, B, C, **D, E, F, G**, D, E, F, G,... Os estados A,B,C são o transiente de tamanho 3 e o período terá tamanho 4, sequência de estados D à G.

Para cada passo de simulação é preciso ter uma cópia do estado atual do grafo no tempo t. Para cada vértice é necessário percorrer todos os seus vizinhos para calcular o novo estado no tempo t+1. Seja N o número de vértices e A o número de arestas. Para cada passo, todas as arestas devem ser visitadas e a função de cada vértice deve ser calculada, resultando em uma complexidade O(N+A) na versão sequencial do algoritmo. Se o grafo tem um transiente T e um período P, a complexidade da simulação será O((T+P)*(N+A)).

Neste trabalho propomos implementar cada vértice diretamente em hardware. O vértice estará conectado aos seus vizinhos, e de maneira síncrona, receberá os valores dos vizinhos. A atualização do novo estado de todos os vértices será realizada em paralelo, reduzindo a complexidade do cálculo de um passo de simulação para O(1).

4. Arquitetura Proposta

Trabalhos anteriores já propuseram o uso de FPGAs como aceleradores para implementação de grafos [8] ou de computação com o modelo de automatos celulares [9]. O ponto principal é o mapeamento da computação paralela no FPGA. Aceleradores em FPGA para redes reguladoras foram propostos em [10,11,12]. Um FPGA com uma arquitetura analógica/digital é apresentado em [10], porém é limitado à 20 genes e um novo FPGA específico com tecnologia mista deve ser fabricado. O acelerador, proposto em [11], usa aprendizado bayesiano com redes probabilísticas, porém fica limitado à redes com 10 genes. O acelerador proposto em [12] é o mais próximo deste trabalho, e também busca o cálculo do período. Entretanto usa uma abordagem com um processador fortemente acoplado ao FPGA, trabalhando em conjunto no cálculo do período. Os resultados de tempo de execução são apresentados apenas para um grafo com 3 genes. Em termos de área, a rede chegar até 800 genes em um FPGA Virtex2. Porém cada vértice pode ter no máximo 5 vizinhos, os vizinhos são determinados em tempo de síntese. O tempo de síntese não é considerado na comparação. Além disso, a abordagem proposta [11] fica restrita a períodos pequenos, devido limitações do modelo.

Este trabalho difere dos anteriores em vários aspectos: apresenta o uso de grafos heterogêneos, modela dinamicamente grafos livre de escala que são mais próximos da biologia [3], além de mostrar que o tempo de execução para grafos com 200 genes é da ordem de microsegundos.

Uma primeira solução para o cálculo do período seria mapear os vértices e arestas diretamente no FPGA. Entretanto, os modelos baseados em simulação necessitam gerar vários grafos para ter uma amostragem significativa. Para cada grafo, pode-se variar as funções locais, estado inicial e as arestas. Neste caso, o tempo de síntese e mapeamento para cada novo grafo no FPGA, irá dominar o tempo de execução. Mesmo para grafos pequenos, o tempo de síntese é superior ao tempo de simulação em software. Além disso, os grafos livre de escala não são regulares e possuem alguns vértices com muitas arestas. Por exemplo, para um grafo com N=100, que foi simulado neste trabalho, possui um vértice 99 com arestas. Isso gera vértices com muitas arestas e roteamento mais complexo.

Este trabalho propõe uma arquitetura genérica onde os vértices são mapeados no FPGA e as arestas são conectadas através de uma rede reconfigurável, ou seja, um nível de reconfiguração acima do FPGA. Para gerar um novo grafo, basta reprogramar rede.

Cada vértice possui uma máquina de estados local para executar o algoritmo descrito na seção 3. O código VHDL do vértice é parametrizado e gerado automaticamente a partir da definição do número de arestas conectadas ao vértice. Primeiro o vértice, recebe os valores dos vértices vizinhos, e executa um passo da simulação S_0. Depois o vértice passa a receber os valores dos vizinhos para a simulação S_1, onde dois passos são executados. Então os valores de S_0 e S_1 são comparados localmente e enviados para um unidade central que verifica se S_0=S_1 para todos os vértices. Quando S_0=S_1, todos os vértices passam a simular com S_1 estacionário, avançando passo a passo S_0, até S_0 ficar igual a S_1. Quando S_0=S_1 novamente, o período é determinado. Este algoritmo é implementado na máquina de estado local de cada vértice. A área no FPGA ocupada pelo vértice varia de 45 à 582 LUTs para um vértices com 2 à 16 vizinhos. O espaço ocupado pelos vértices em um FPGA atual mostra que é viável simular grafos com tamanhos entre 100 e 2000, que estão próximos dos valores usados na biologia [2]. Este espaço pode ser reduzido se usarmos funções de transições mais simples com operadores And/Or ou funções canalizadores que são as funções mais usadas nas implementações por simulação [1,4,5,6]. Neste trabalho usamos a função majoritária que envolve uma operação de soma, que tem custo superior em área que expressões canalizadores com inibidores e ativadores And/Or.

Para ter flexibilidade, sem a necessidade de resintetizar, um conjunto genérico de vértices é gerado e sintetizado para um dado valor de N. A maioria dos vértices tem de um à dois vizinhos. Alguns poucos vértices são gerados com um número maior de vizinhos. Para determinar a quantidade de cada tipo de vértice, vários grafos livre de escala foram gerados, um histograma do número de arestas por vértice foi construído para gerar um modelo com as características dos grafos livre de escala.

Um grafo livre de escala é gerado por uma função de probabilidade $P(k)=Ck^{-\gamma}$. Esta função é responsável por distribuir as arestas para cada vértice do grafo, para maiores detalhes consultar [4,6]. Dependendo dos valores de γ e k, o número médio e máximo de arestas podem ser bem elevado. Por exemplo, um grafo pode ter 10 a 15 vizinhos em média por vértice e alguns vértices estarão ligados a todos os outros. A maioria terá poucos vizinhos. A solução proposta aqui é limitar o número máximo de conexões de um vértice. Se o número de vizinhos é maior que o número de conexões do vértice, será necessário realizar em vários passos a transmissão de todos os valores dos vértices vizinhos. Suponha que um vértice tenha 10 vizinhos. Suponha que ele tenha sido mapeado em um vértice com 3 conexões. Serão necessários pelo menos 4 transmissões para que o vértice receba os valores dos vizinhos e recalcule seu estado.

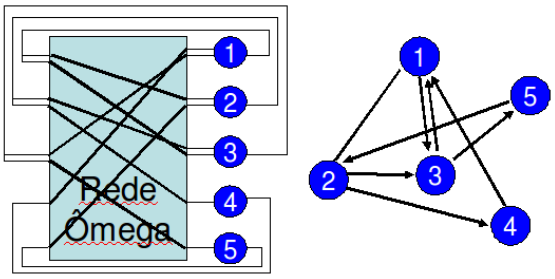


Figura 2. Um grafo e sua implementação com a rede multiestágio Omega

Para gerar uma solução flexível e de baixo custo em área sem perder em desempenho, este trabalho propõe o uso de uma rede multiestágio como rede de conexão global entre os vértices. A Fig. 2 ilustra o exemplo de um grafo com 5 vértices. Os vértices são conectados a uma rede multiestágio. A solução adotada foi uma rede Omega [7]. O mapeamento é gerado em software a partir de um grafo livre de escala. Para cada distribuição diferente de arestas, o número mínimo de configurações necessárias para realizar todas as conexões é gerado. A Fig. 3 ilustra um outro grafo com 5 vértices mas com um padrão diferente de ligação. A rede Omega é reconfigurada e uma nova simulação pode ser executada. A reconfiguração dinâmica da rede para gerar uma nova simulação permite um alto ganho de desempenho na execução da simulação.

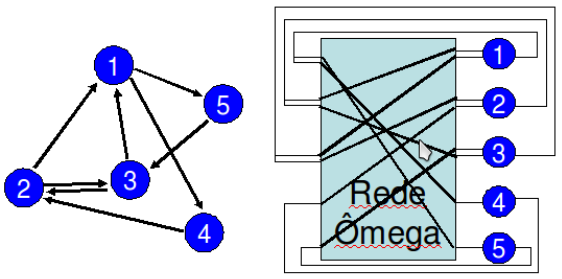


Figura 3. Um novo grafo com a rede multiestágio Omega reconfigurada

As redes multiestágios com N entradas tem custo O(N Log N), que é significativamente menor que o custo $O(N^2)$ das redes crossbar. O atraso é O(log N). Além disso, a síntese em FPGA se mostrou eficiente.

5. Resultados e Conclusões

Para avaliar o desempenho, um experimento com dois grafos livre de escala com N=100 e N=200 foi realizado. O tempo de execução no FPGA foi simulado em uma Virtex5 à 100Mhz. Os valores dos estados iniciais foram os mesmos para a versão em software e a versão em FPGA. Os valores dos estados durante a simulação das duas versões foram comparados e validados.

O processo de geração do grafo foi feito em software. Primeiro, dado um valor de N, uma arquitetura é gerado para o FPGA. Este passo é feito apenas uma vez. Para cada simulação, um novo grafo será gerado com um conjunto diferentes de arestas, os vértices são mapeados na arquitetura e roteados na rede multiestágio. A configuração do roteamento é carregado no FPGA. Para cada grafo, vários estados iniciais serão simulados. Cada simulação retorna o período daquela instância do grafo.

A tabela 1 mostra os resultados. A versão em FPGA, devido a execução em paralelo e a comunicação eficiente pela rede multiestágio, mesmo usando mais de um passo devido ao alto grau de alguns vértices, gerou uma aceleração de 3 ordens de grandeza. A implementação em software foi feita em C++ e executada em um computador com 2.2Ghz clock.

N	Tempo Soft	Tempo FPGA	Aceleração
100	16 ms	23 us	700
200	100ms	86us	1150

Table 1. Tempo de simulação e Aceleração

O grafo com N=100, possui 933 arestas, e foi implementado em uma arquitetura com 12 vértices com 16 conexões, 11 com 8 conexões, 28 com 4 e 49 com duas conexões. A rede possui 512 conexões, que ocupou 4608 LUTs no FPGA. O período encontrado foi de 26 estados, e o transiente de 46. Vale ressaltar que para grafos de tamanho 100, pode-se encontrar períodos de tamanho acima que 8.000. Neste caso, a solução em FPGA apresentará um ganho significativo em tempo. O grafo com N=200 foi implementado com uma rede Omega com 1024 entradas e possui 2331 arestas. Uma rede Omega de 1024 ocupou 10752 LUTs no FPGA.

Apesar do tempo de simulação de uma instância ser muito pequeno, mesmo para versão em software, da ordem de milisegundos, este tempo depende muito do período e do transiente. No processo de simulação para o estudo do comportamento das redes, são gerados 10^5 ou mais instâncias de grafos (variando as conexões, função de transição ou simplesmente o estado inicial) [1]. Portanto é fundamental ter um tempo reduzido. Os trabalhos anteriores limitam o estudo a pequenos grafos com N=20 em [6], N=50 em [4], N=40 com 10^3 instâncias em [1] e apenas 200 amostras para N=400 em [1], devido ao tempo de simulação. Este trabalho apresenta uma solução que permite a exploração de um número mais significativo de simulações.

Este trabalho teve como objetivo validar a simulação paralela com a máquina de estados nos vértices e uso da rede multiestágio, mostrando o potencial da solução. Trabalhos futuros incluem a implementação da solução em uma placa aceleradora e simulações com um número significativo de instâncias.

Referências

[1] K. Iguchi, S. Kinoshita, H. S. Yamada, “Boolean dynamics of Kauffman models with a scale-free network”, Journal of Theoretical Biology, vol 247, pp 138-51, 2007.

[2] A. Garg, A. Di Cara, I Xenarios, L. Mendoza, G. De Micheli, “Synchronous versus asynchronous modeling of gene regulatory networks”, BIOINFORMATICS, V 24(17), pp 1917-1925, 2008.

[3] A. Barabási, Z. N. Oltvai, “Network biology: understanding the cell's functional organization”, Nature Review Genetics, Vol 5, pp 101-113, 2004.

[4] D. J. Irons, “Improving the efficiency of attractor cycle identification in Boolean networks”, Physica D:, Vol 217, Issue 1, pp 7-21, 2006.

[5] A. Bhattacharjya, S. Liang, “Median attractor and transients in random boolean nets”, Physica D: Nonlinear Phenomena, vol 95, pp 29-34, 1996.

[6] M. Aldana, “Boolean dynamics of networks with scale-free topology”, Physica D,185(1), 45-66, 2003.

[7] D.H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. Comp.*, Vol. C-24, December 1975, pp. 1145-1155.

[8] Lorenz Huelsbergen, "A Representation for Dynamic Graphs in Reconfigurable Hardware and its Application to Fundamental Graph Algorithms," International ACM FPGA, pp.105-115, 2000.

[9] Kobori, T., Maruyama, T., and Hoshino, T. 2001. A Cellular Automata System with FPGA. In *Proceedings of IEEE FCCM* (2001). pp120-129.

[10] Tagkopoulos, I., Zukowski, C., Cavelier, G., and Anastassiou, D. A custom FPGA for the simulation of gene regulatory networks. In *Proceedings of ACM Great Lakes Symposium on VLSI, 2003*

[11] I. Pournara, C.S. Bouganis, and G.A. Constantinides. Fpga-accelerated bayesian learning for reconstruction of gene regulatory networks. In *Proceedings of IEEE FPL 2005,* pages 323--328, 2005

[12] Zerarka, M.T. David, J.P. Aboulhamid, E.M. High speed emulation of gene regulatory networks using FPGAs. In *Proceedings of IEEE 47th Midwest Symposium on Circuits and Systems*, 2004

Sessão Regular 4

Arquitectura e Circuitos Aritméticos

Moderação: Iouliia Skliarova
Universidade de Aveiro / IEETA

A distributed cache memory system for custom vector processors*

João M. Meixedo and José C. Alves
{*jmeixedo@inescporto.pt, jca@fe.up.pt*}
FEUP / INESC-Porto

Abstract

This paper presents a parameterized distributed cache memory system for application specific processors implemented in FPGA devices. The system is made of several direct mapped cache memory modules that share the access to a single external data memory, and provide parallel data lanes that will feed the inputs of an arithmetic datapath. Each cache block is assigned to one or more application data vectors and includes a module to compute the effective memory address of each data value (32 bit), based on a reduced set of 4-bit commands that specify the iterations over up to 3 vector indexes. A prototype memory system was implemented and verified on a Virtex4LX80-10 FPGA, supporting one cycle reading latency of data located in the cache memory and a clock frequency of 200 MHz.

1. Introduction

Application specific vector processors can be an effective mean to improve the performance of conventional (scalar) processors. This is particularly interesting for embedded applications implemented in field-reconfigurable devices with integrated processors, where important gains in speed can be leveraged by custom designed deep pipelined datapaths to handle sequences of computations on vectors of data. Current FPGA devices can effectively host pipelines with tens of floating-point arithmetic operators, reaching performances up to a few giga flops. However, feeding the required data to minimize (ideally avoid) pipeline stalls can be impossible without the support for an adequate bandwidth to the data memory. This is the usual situation in FPGA-based systems where the main data memory is implemented by low cost dynamic memories that exhibit long reading latencies.

Vector architectures implementing the SIMD paradigm are being used for years to execute efficiently computing applications that perform operations on vectors of data. A vector processor extends the datapath of a conventional scalar CPU by including additional memories that form a vector register file, along with vector instructions that apply to the whole set of elements of the vector operands. Important performance gains can be achieved by building complex vector instructions that push their operands (vectors) through a pipelined datapath built by chaining arithmetic operators, as can be commonly identified in various sections of an application.

With current FPGAs it is possible to create deep pipelines with several floating point arithmetic operators and input operands. In spite of the high performance potential attained by such pipelines, to effectively use them it is necessary a convenient memory organization that may be able to provide enough data bandwidth to the datapath inputs. The ideal (and obvious) solution is to use dedicated memory banks to implement independent register files. However, limitations on the quantity of inter-chip memory available and the practical impossibility of populating discrete memories off-chip makes this approach usable only when the number and size of vectors used by an application is compatible with the quantity of memory that may be allocated to the vector registers.

In this paper we propose a parameterized and distributed cache memory system aimed to be implemented within a FPGA device, including dedicated but simple address generators for vector applications. The rest of the paper is organized as follows. Section 2 summarizes works of other authors related to the main subject of this paper. In section 3 a general overview of the memory system is presented. Sections 4 and 5 describe the architecture of the parameterized cache and the vector address generator associated with each cache block, respectively. Finally, section 6 summarizes the preliminary results and concludes the paper presenting plans for future developments.

2. Related work

Vector processing is being used for several years in high end processors and supercomputers to effectively exploit the data-level parallelism observed in many computing applications [1]. Until the appearance of high-density FPGAs by the late 90's, vector processing was an exclusive feature of commercial high-performance processors, application oriented processors like DSPs or GPUs or sophisticated custom designed machines.

Current FPGAs that include several inter-chip arithmetic functions and memory blocks offer now a technology capable of supporting practical applications of custom vector processing as a mean to meet the performance requirements of demanding embedded applications. This has motivated the development of vector processing units for embedded applications that act as auxiliary processors of conventional CPUs. Making use of hardware customization, the specific needs of a problem (eg. num-

*This work is funded by FCT (*Fundação para a Ciência e Tecnologia*), project PTDC/EEA-ELC/71556/2006

ber of processing lanes or organization of vector register file) can be exploited to better utilize the limited hardware resources of FPGA devices. Customizable and scalable vector FPGA-based co-processors were proposed in recent works [2, 3, 4], as a means to increase the computing power of embedded systems based on on-chip soft processors, like the MicroBlaze or the NIOS-II.

Targeting CMOS (non-configurable) technology, the VIRAM architecture [5, 6] developed at the University of California at Berkeley, USA, is a scalable vector co-processor for the 64-bit MIPS core that implements a multi lane processing core with a centralized vector register file, aimed for multimedia applications. A different microarchitecture from the same authors CODE [7] introduces a clustered vector register file that distributes the vector registers defined in the ISA by different (physical) groups, thus reducing the data traffic among functional units.

Memory access bandwidth is a key issue that affects significantly the performance of vector processors. The gains in speed obtained by processing vectors of data can only be effective when the memory system is capable of providing the required operands to the arithmetic units as close as possible to the fastest rate allowed by the datapath, thus avoiding pipeline stalls. Because it is not practical, mainly for cost reasons, to attach to a FPGA-based processing system lots of fast off-chip memory chips, the constraints imposed by the limited amount of inter-chip memory blocks in FPGA devices do require a careful design of the whole memory system.

With the relatively low granularity of memory blocks available in modern FPGAs, it is easy to organize different configurations of the memory system, with respect to the number of blocks, their depth and width. When the application data can be held entirely in the internal RAM blocks, the memory system may be organized in order to allocate sets of variables (either scalars or vectors) to several independent memories that can be accessed in parallel to feed the inputs of multi-operand datapaths at clock rate. This was exploited in [8] with a set of thirteen, 16 KByte dual-port memories, each one holding a $16 \times 16 \times 16$ 3D matrix and feeding at clock rate the inputs of a deep pipeline with 15 floating point arithmetic operators.

When external memories are needed to hold large data sets, the slow access may compromise the efficiency of the execution datapath, unless appropriate memory caching mechanisms are used to exploit the temporal and spatial locality of data. The utilization of cache memory and data prefetching for FPGA-based vector processors has been addressed in [9], where the authors study the design trade-offs for different data cache organization in a soft vector processor, while optimizing the utilization of the internal FPGA blocks of RAM. Data prefetching was exploited in order to deal with the burst access modes of modern dynamic memories, while trying to avoid filling the cache memories with surplus data.

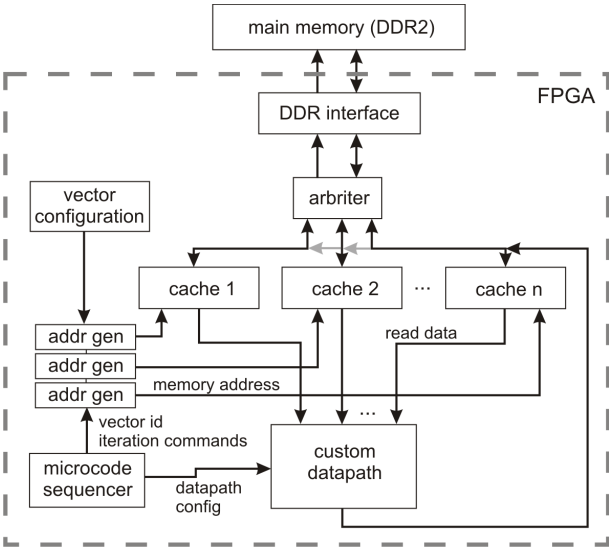


Figure 1. General organization of the cache memory system.

3. Parameterized cache memory system

In this work we extend the proposal of automatic cache generation for FPGAs [10] to build a cache memory system for vector processors, using a set of independent cache memories built with the internal SRAM block memories present in modern FPGA devices. The data width is 32 bits (for single precision floats) and each cache memory bank can be configured with different cache line size and depth. For now, only direct-mapped cache memories are supported and the whole design has been specialized for a specific family of FPGAs (Xilinx Virtex4). Besides, only 1D, 2D and 3D vectors can be handled by the address generation unit, with their elements residing in contiguous memory positions, line-by-line (for 2D and 3D vectors). This memory system is intended to implement the interface between an external dynamic memory and a custom vector processor, providing, in parallel, several data values to a custom designed pipelined datapath.

Presently this has been integrated with a simple microcode controller that issues sequences of reading commands from data vectors allocated to 4 different cache blocks. The whole system has been implemented in a Virtex4 LX80 FPGA connected to a 512 MB DDR2 memory module, integrated in a DN8000K10PSX prototyping board from the Dini Group company (www.dinigroup.com).

Figure 1 illustrates the general organization of the system and implementation details are presented in the next sections.

4. Cache memories

The configuration of each cache memory block is specified by the parameterization of a Verilog synthesizable model. Although this model do not explicitly instantiate any XILINX-specific primitives, the Verilog templates used to code the blocks of RAM memory are specific of

the Xilinx synthesis tool and may not map to similar RAM blocks present in different FPGA technologies or when using other synthesis tools.

Because the primitive SRAM blocks in Virtex4 FPGAs are 18 Kbit, the size of each cache memory must always be a multiple of 2 KByte (16 Kbit) in order to fully utilize the block memories allocated. Also, because a reading command from the DDR2 memory always returns a 32 byte block in two consecutive clock cycles (128+128 bits), the cache line size must be always defined in multiples of 32 bytes.

The associative memory was designed to be mapped into distributed memory built with lookup-tables and flip-flops, in order to reduce the read cycle (when cache hit) and the write cycle (cache miss) to a single clock period. A simple cache line replacement policy was implemented, that always substitutes the oldest written cache line. This was implemented using a FIFO for the associative memory and simple arithmetic to map each entry of the associative memory to the cache block that actually holds the data.

Two additional replacement policies can be chosen that share similar resources: LRU (least recently used) and LFU (least frequently used). A set of history registers associated with each entry of the associative memory represent either the aging of a cache line or the frequency of reading from that line, depending on the replacement policy selected.

To implement LRU, a read hit from cache line i sets its history register HR_i to the maximum value (all ones) and decrement all the registers associated to the other lines by one unit (the same happens when cache line i is replaced with new data). This is only done if HR_i has not yet the maximum value, meaning that the previous read operation was not issued from the same cache line. This avoids that repeated reads from the same cache line rapidly decrement the aging registers assigned to the other cache lines. The entry of the associative memory to be written when a replacement occurs is determined by the current values in the history registers, selecting the lowest value (meaning the oldest accessed cache line). Because the effective write into the associative memory only needs to be done when the data requested effectively arrives from the main memory, the calculation of the minimum among all the history registers can be done sequentially, within a time budget equal to the read latency of the DDR2 memory (22 clock cycles in the current implementation). Figure 2 details the logic circuit that implements the update of the history registers for LRU.

To implement the LFU technique, the selection of the cache line to be replaced is also done by choosing the cache line which history register has the minimum value. In this case, the set of history registers build a histogram representing the frequency of read accesses from each line. When a cache hit occurs and the history register HR_i of line i still does not have the maximum value, it is incremented by one; if current value is the maximum, then all the values in the history registers are divided by 2. Figure 3 presents the logic circuit that implements this mechanism.

The access to the main memory is shared by all the cache blocks instantiated in the memory system. A control module manages the read aand write requests issued from the different cache blocks and performs the reading operations, according to predefined priorities assigned to each cache block.

5. Address generator

Each cache memory block is assigned to one or more data vectors whose dimensions and locations in memory (absolute address) are known at synthesis time. Associated to each cache block, a dedicated address generator converts references to elements in a vector (the requested element indexes, for vectors up to 3 dimensions) into the absolute memory address that is then sent to the cache block. Instead of referencing absolute indexes, what would require additional arithmetic to compute the effective memory address, the references to vector elements are encoded into a small set of commands that specify an iteration over the previous reference (for example `A[i++,j]`). This translates to simple loads, additions and subtractions of constants to the address register and reduces significantly the number of control lines necessary from the microinstruction.

Table 1 presents the iteration commands implemented and the operations required to calculate the absolute memory address. Label ADDR represent the address of the

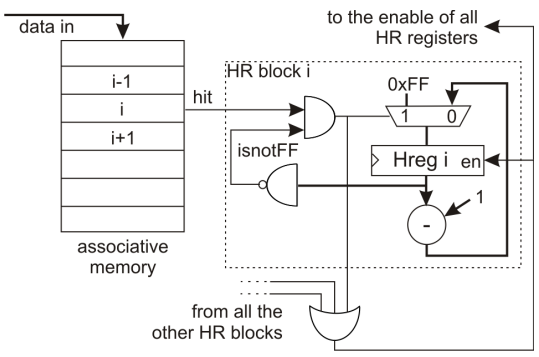


Figure 2. Logic circuit of the history registers (HR) for implementing the LRU replacement policy.

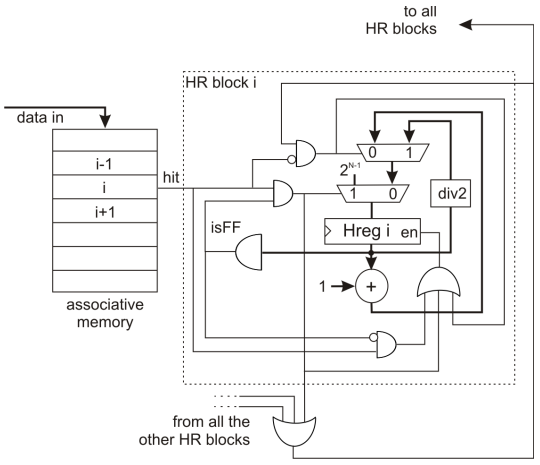


Figure 3. Logic circuit of the history registers for implementing the LFU replacement policy.

Iteration	memory address
A[i++,j,k]	ADDR+1
A[i--,j,k]	ADDR-1
A[i,j++,k]	ADDR+NI
A[i,j--,k]	ADDR-NI
A[i,j,k++]	ADDR+NI*NJ
A[i,j,k--]	ADDR-NI*NJ
A[0,0,0]	START
A[0,j,k]	START_I
A[i,0,k]	START_J
A[i,j,0]	START_K
A[NI-1,j,k]	START_I+NJ-1
A[i,NJ-1,k]	START_J+NJ*NI-NJ
A[i,j,NK-1]	START_K+NJ*NI*NK-NI*NJ

Table 1. Example of iteration commands implemented by the address generator. This considers a 3D vector A[,,] located in the main memory at address START and with NI, NJ and NK elements along each of the 3 dimensions.

last element accessed (a register) and START is a constant that represents the memory address of the first element in the vector. Three additional registers (START_I, START_J, START_K) are maintained with the address of the first element of a row along each dimension.

6. Results and conclusions

In this paper we proposed a parameterized cache memory system, aimed to increase the effective memory bandwidth for vector applications, while making use of the fast block RAMs present in modern FPGA devices. This will be later integrated into a design framework to automate the synthesis of application specific vector processors.

A first implementation was done to a Virtex4LX80-10 FPGA, including 4 independent cache blocks with LRU replacement policy. The writing process implementing the write-allocate policy has been validated in simulation but it was not yet integrated in a real hardware implementation. To issue a series of reading commands, a simple microcode sequencer sends to the cache memories a sequence of the iteration commands presented in table 1. With 4 cache memories, each one with 32 lines and 16 Kbit per line (for a total of 2 Mbit of RAM), the design uses 68% of the BRAMs, 7% of LUTs and 4% of flip-flops. This design has been sucessfully verified with a 200 MHz clock, which is the maximum frequency allowed by the interface used to access the external dynamic memories.

References

[1] Mateo Valero Roger Espasa and James E. Smith. Vector architectures: Past, present and future. In *Proceedings of the 2nd Intl. Conference on Super Computing*, pages 425–432, July 1998.

[2] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. Vespa: portable, scalable, and flexible fpga-based vector processors. In *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 61–70, New York, NY, USA, 2008. ACM.

[3] Jason Yu, Guy Lemieux, and Christpher Eagleston. Vector processing as a soft-core cpu accelerator. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 222–232, New York, NY, USA, 2008. ACM.

[4] Junho Cho, Hoseok Chang, and Wonyong Sung. An fpga based simd processor with a vector memory unit. In *Proc. IEEE International Symposium on Circuits and Systems ISCAS 2006*, pages 4 pp.–, 2006.

[5] Christoforos Kozyrakis. *Scalable Vector Media-processors for Embedded Systems*. PhD thesis, Computer Science Division, University of California, Berkeley, May 2002.

[6] D.A. Kozyrakis, C.E. Patterson. Scalable, vector processors for embedded systems. *Micro, IEEE*, 23(6):36–45, Dec. 2003.

[7] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. In *Proc. 30th Annual International Symposium on Computer Architecture*, pages 399–409, 2003.

[8] Filipe Oliveira, C. Silva Santos, F. A. Castro, and José C. Alves. A custom processor for a TDMA solver in a CFD application. In *ARC '08: Proceedings of the 4th international workshop on Reconfigurable Computing*, pages 63–74, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] J. Gregory Steffan Peter Yiannacouras and Jonathan Rose. Improving memory system performance for soft vector processors. In *WoSPS: Workshop on Soft Processor Systems*, 2008.

[10] P. Yiannacouras and J. Rose. A parameterized automatic cache generator for fpgas. In *Proc. IEEE International Conference on Field-Programmable Technology (FPT)*, pages 324–327, 2003.

Divisor Decimal em FPGA com o Método de Newton-Raphson

Pedro Pereira[†], Mário Véstias[†], Horácio Neto[‡]
[†]*INESC-ID/ISEL/IPL,* [‡]*INESC-ID/IST/UTL*
24624@alunos.isel.ipl.pt, mvestias@deetc.isel.ipl.pt, hcn@inesc-id.pt

Resumo

A norma IEEE 754-2008 inclui a especificação dos formatos e da aritmética decimal. O cálculo aritmético em decimal tem a vantagem de não introduzir erros de conversão para binário sempre que os dados se encontram representados em decimal. No entanto, como, em geral, não existe suporte hardware para o cálculo decimal, recorre-se a rotinas de software para efectuar as operações directamente em aritmética decimal. A desvantagem é que quando executado em software, o cálculo decimal tornam-se demasiado lento relativamente ao cálculo binário. Com o objectivo de acelerar os cálculos aritméticos decimais, foram propostas unidades aritméticas hardware dedicadas, incluindo o somador, o multiplicador e o divisor. Este artigo descreve a implementação de um divisor decimal em hardware reconfigurável.

1. Introdução

A aritmética binária é o método mais utilizado no cálculo aritmético devido à sua simplicidade quando comparado com o cálculo decimal. No entanto, recentemente, o cálculo decimal é tido como essencial sempre que se tratam de aplicações financeiras ou comerciais, em que os operandos estão maioritariamente em representação decimal e não são admitidos erros nos cálculos quando comparados com os resultados obtidos por cálculo manual.

As implementações software da aritmética decimal são demasiado lentas, cerca de 3 a 4 ordens de magnitude mais lentas quando comparadas com a aritmética binária em hardware.

Certos processadores, como o IBM Power6 [1], já incluem unidades hardware dedicadas para o cálculo decimal com vírgula flutuante. As unidades incluem somadores, subtractores, multiplicadores e divisores. A multiplicação decimal é bastante mais complexa que a multiplicação binária devido à dificuldade associada em representar números decimais em binário. O mesmo se passa com a divisão, embora se possam utilizar alguns dos algoritmos utilizados na implementação da divisão binária.

As operações decimais podem ser realizadas através da manipulação directa dos números em decimal recorrendo a métodos iterativos [2], [3], [4], em que os dígitos do resultado vão sendo gerados iterativamente ou em paralelo (em que o resultado pode ser gerado num único ciclo de relógio). Em alternativa, as operações decimais podem ser realizadas com aritmética binária, bastando para tal converter entre binário e decimal. Por exemplo, quando a tecnologia alvo são as FPGA, este método permite tirar proveito dos multiplicadores e dos somadores embebidos [5]. Um dos grandes problemas associados a este método é o da conversão de binário para BCD.

Neste artigo, propomos uma arquitectura para um divisor decimal de 7 dígitos com base em aritmética binária. A abordagem baseia-se na utilização do método de Newton-Raphson para o cálculo da divisão e recorre a multiplicadores cuja implementação se baseia na arquitectura de multiplicadores decimais proposta em [5]. As implementações existentes de divisores decimais baseiam-se na divisão iterativa com o método de Newton-Raphson [6] e no método de divisão por recorrência de dígito [7], [8], [9] e [10]. A proposta apresentada neste artigo também usa o método de Newton-Raphson, mas com uma aproximação inicial mais eficiente.

A secção 2 apresenta a arquitectura dos multiplicadores decimais. Na secção 3, descreve-se o algoritmo considerado para realizar a divisão decimal. Na secção 4, apresenta-se a arquitectura do divisor decimal, de acordo com o algoritmo da secção 3. Na secção 5, apresentam-se os resultados de implementação do divisor decimal. Finalmente, na secção 6, terminamos o artigo com as conclusões e o trabalho futuro.

2. Multiplicação Decimal

Na implementação dos multiplicadores decimais necessários à implementação do divisor, adoptou-se a solução apresentada em [5]. Os operandos são convertidos de decimal para binário, é feito o cálculo em binário e o resultado é convertido de novo para decimal. A referência indicada apenas considera a multiplicação de operandos com até 5 dígitos (ver figura 1). Para multiplicações de maior dimensão, podem-se considerar pelos menos duas opções:

- Aumentar o tamanho das unidades de conversão BCDtoBIN e BINtoBCD;
- Utilizar produtos parciais.

Para exemplificar os dois métodos, consideremos um multiplicador decimal de 8×8 dígitos decimais.

Considerando uma implementação sem produtos decimais parciais, teríamos a configuração da figura 1.

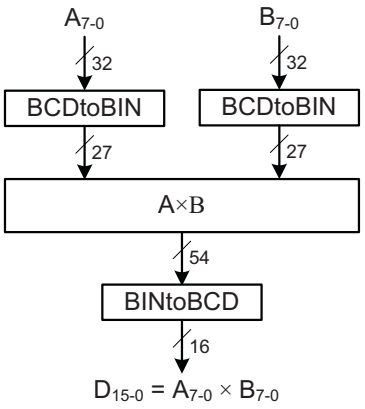


Figura 1. Multiplicação decimal 8×8 sem produtos decimais parciais

Na implementação dos conversores são consideradas as implementações apresentadas em [5], cujos resultados se apresentam na tabela 1.

Multiplicador	Área (LUT)	Atraso (ns)
BCD2BIN	174	10
BIN2BCD	2242	18
MULT27x27	797	13
Total	3213	41

Tabela 1. Resultados do multiplicador

Se por outro lado, considerássemos uma implementação com produtos parciais decimais, teríamos, por exemplo, a arquitectura da figura 2, em que os operandos são divididos em dois grupos de quatro dígitos cada.

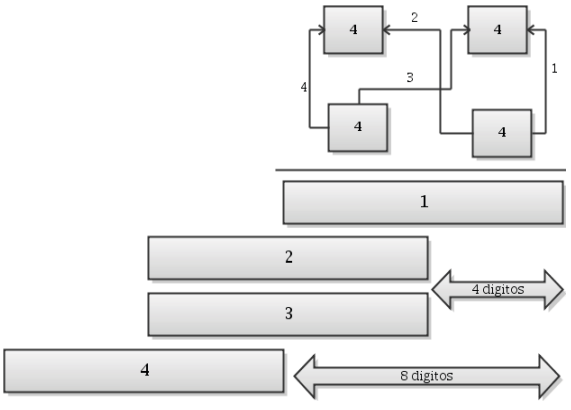


Figura 2. Multiplicação decimal 8×8 com produtos decimais parciais

Os produtos parciais 2 e 3 da figura estão alinhados um com o outro, pelo que podem ser somados em binário e posteriormente convertidos para BCD. O resultado é somado com as restantes parcelas, anteriormente convertidas em BCD. Os multiplicadores utilizados no divisor seguem uma estrutura idêntica ao da figura 1, mas em que os operandos têm metade da dimensão. Para esta arquitectura, temos os resultados da tabela 2.

Multiplicador	Área (LUT)	DSP	Atraso (ns)
Total	2049	0	22
Total	1239	1	10

Tabela 2. Resultados do multiplicador decimal 8×8 com produtos decimais parciais

A diferença na utilização de recursos e as frequências de operação tendem a piorar com o aumento do número de dígitos dos operandos. Como tal, optou-se por usar os multiplicadores com produtos parciais para realizar as multiplicações do divisor.

3. Divisão Decimal

3.1. Algoritmo para a Divisão Decimal

O cálculo da divisão baseia-se no cálculo do recíproco com base no método iterativo de Newton-Raphson. Para calcular x = 1/d, usamos o método de Newton-Raphson para determinar a raiz de f(x) = 1/x - d. O método consiste no cálculo iterativo da seguinte equação.

$$x^{i+1} = x^i(2 - x^i \times d) \quad (1)$$

Em cada iteração são necessárias duas multiplicações e o cálculo do complemento para dois. A convergência do método é quadrática, ou seja, por cada iteração do método, o número de dígitos correctos duplica.

3.2. Aproximação MiniMax

Para reduzir o número de iterações, usa-se uma tabela com uma aproximação inicial do recíproco, ou seja, usa-se uma função polinomial de aproximação a 1/x, em que os coeficientes são guardados numa tabela e usados no cálculo do polinómio para gerar a aproximação inicial. O número de dígitos correctos obtidos com esta aproximação depende do método usado na aproximação e do grau do polinómio. Neste trabalho, usou-se um polinómio de 1ª ordem do tipo $y_0 + m \times (x - x_0)$ e o método de aproximação MiniMax. Um polinómio de 1ª ordem garante uma aproximação inicial com precisão suficiente para que seja necessária apenas uma iteração do método de Netwon-Raphson para obter 7 dígitos de precisão. Um polinómio de segundo grau permite gerar uma aproximação inicial mais precisa, mas não garante a precisão inicial de 7 dígitos, pelo que também seria necessária uma iteração do Netwon-Raphson. Consequentemente, a complexidade em termos de hardware é maior. Para polinómios de grau superior, a complexidade no cálculo do polinómio é superior à usada com uma aproximação de 1ª ordem.

Utilizando um polinómio de 1º grau, os coeficientes são dados pelas equações (2-3).

978-972-789-304-1 REC'2010

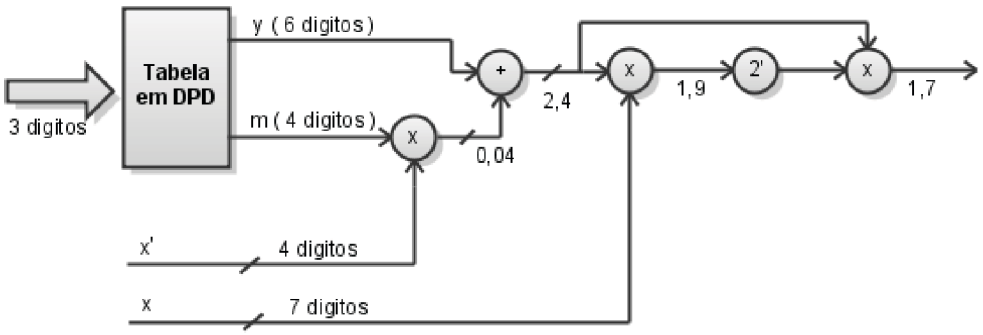


Figura 3. Arquitectura do divisor decimal

$$y_0 = \frac{1}{2x_0} + \frac{1}{2x_1} + \frac{1}{\sqrt{x_1 x_0}} \quad (2)$$

$$m = -\frac{1}{x_1 x_0} \quad (3)$$

O erro máximo do método para um polinómio de 1º grau é dado pela equação (4).

$$E_{max} = \frac{1}{x_0} + \frac{1}{x_1} - \frac{2}{\sqrt{x_1 x_0}} \quad (4)$$

O erro depende dos pontos inicial e final do intervalo da aproximação. Para reduzir o erro, considera-se o intervalo inicial [0.1,1[subdividido em pequenos sub-intervalos e aplica-se o método a cada um dos sub-intervalos. Os coeficientes de cada um dos polinómios de aproximação são depois guardados numa tabela. Quantos mais sub-intervalos se considerarem, melhor será a aproximação. No entanto, maior será a tabela para armazenar um maior número de coeficientes. Para ter uma ideia dos erros associados a diferentes intervalos, consideremos os erros associados ao intervalo de maior erro (intervalo iniciado em 0.1) com diferente número de sub-intervalos (ver tabela 3)

Intervalo	Erro
[0.1, 0.10001]	1,25E-08
[0.1, 0.1001]	1,25E-06
[0.1, 0.101]	1,23E-04
[0.1, 0.11]	1,08E-02

Tabela 3. Erro da aproximação em função do intervalo da aproximação

Naturalmente que quanto menor o intervalo, menor o erro. No entanto, para um erro de, por exemplo, 1.25×10^{-6} são necessários 9000 intervalos. Considerando, por exemplo, 32 bits por cada par de coeficientes, a tabela teria um tamanho de cerca de 35 Kbytes. Para reduzir o tamanho da tabela, no divisor considerado neste artigo, optou-se por considerar intervalos de 0.001, que corresponde a ter 900 intervalos. Considerou-se, ainda, 6 dígitos para o y_0 e 4 dígitos para o m (ver secção 4). O resultado é uma tabela com 900×40 bits).

Para reduzir ainda mais a tabela, os números decimais são guardados em formato DPD (*Densely Packed Decimal*), passando a ter uma tabela de 900×36 bits.

Foi realizado um programa em Python por forma a gerar os valores DPD correspondentes ao y e m obtidos anteriormente pelo método MiniMax (ver primeiros valores na tabela 4). O programa codifica os números da esquerda para a direita por cada grupo de três dígitos, tendo sempre em consideração se o número é separável em blocos de três.

x	y	m
0.100	9,99988	-99.01
0.101	9,90087	-97.07
0.102	9,80381	-95.18
0.103	9,70863	-93.35
0.104	9,61528	-91.58
0.105	9,52370	-89.85
0.106	9,43386	-88.17
0.107	9,34569	-86.54

Tabela 4. Primeiras entradas da tabela DPD

3.3. Codificação e descodificação do formato DPD

A codificação e a descodificação para e de o formato DPD são feitas com circuitos lógicos bastante simples. Durante a compressão, três dígitos BCD (abcd, efgh, ijkm) são codificados em 10 bits (pqr, stu, v, wxy), tendo por base a Tabela 5.

Por exemplo, se o número a codificar for 835 (1000 0011 0101) então a sequência binária "aei"é "100". Esta sequência indica se os números a codificar são pequenos (0 a 7) ou grandes (8 e 9). A compressão fica então "100 011 1 101". De igual forma, o número 009 (0000 0000 1001) tem como resultado da codificação a sequência binária "000 000 1 001". A posição e a escolha dos bits indicativos (v,wx) permite que todos os números de um só dígito estejam alinhados à direita como se fossem codificados em BCD. Na prática, todos os números entre 0 e 79 têm essa característica.

Na descodificação, os 10 bits (pqr, stu, v, wxy) são descodificados para formar os três dígitos BCD (abcd, efgh, ijkm), com base na Tabela 6.

A implementação da codificação e da descodificação é concretizada com recurso a simples operações logicas. Por exemplo, na codificação, p=b+aj+afi e na descodificação $a = vw(\bar{s}+t+\bar{x})$.

Quando implementado em FPGA, o atraso é de apenas

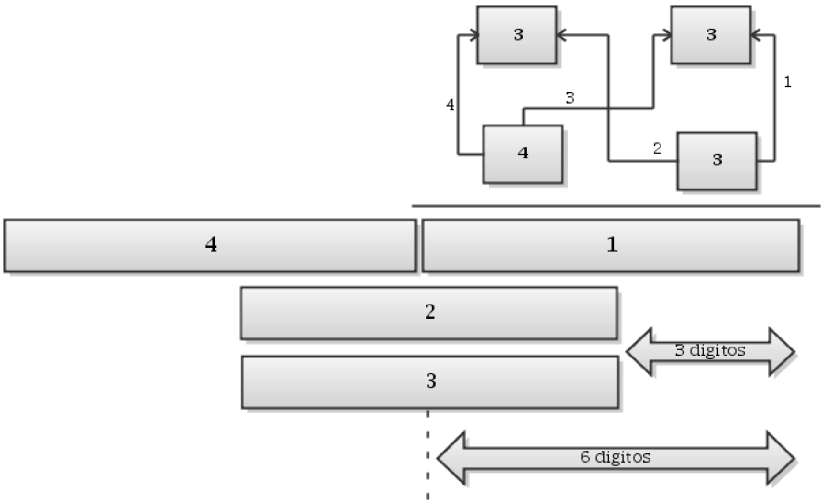


Figura 4. Multiplicação Decimal 6×7

aei	pqr stu v wxy	Observações
000	bcd fgh 0 jkm	Todos os dígitos são pequenos
001	bcd fgh 1 00m	Dígito à direita é grande [mantém 0 - 9 estático]
010	bcd jkh 1 01m	Dígito do meio grande
011	bcd 10h 1 11m	Dígito à esquerda é pequeno [Meio e Direita são grandes]
100	jkd fgh 1 10m	Dígito da esquerda grande
101	fgd 01h 1 11m	Dígito do meio é pequeno [Esquerda e Direita são grandes]
110	jkd 00h 1 11m	Dígito à direita é pequeno [Esquerda e Meio são grandes]
111	00d 11h 1 11m	Todos os dígitos são grandes (dois bits por utilizar)

Tabela 5. Codificação DPD

vwxst	abcd efgh ijkm
0xxxx	0pqr 0stu 0wxyz
100xx	0pqr 0stu 100y
101xx	0pqr 100u 0sty
110xx	100r 0stu 0pqy
11100	100r 100u 0pqy
11101	100r 0pqu 100y
11110	0pqr 100u 100y
11111	100r 100u 100y

Tabela 6. Descodificação DPD

uma ou duas LUT.

4. Arquitectura do Divisor Decimal

A figura 3 mostra o fluxo de dados do divisor com a indicação do número de dígitos utilizados por cada operação.

Na arquitectura são indicados os números de dígitos considerados após cada operação. As truncagens foram determinadas experimentalmente de modo a garantir 7 dígitos de precisão no final da divisão.

A primeira multiplicação e a soma são usadas no cálculo da aproximação. As restantes operações são usadas no cálculo de uma iteração do método de Newton-Raphson.

De seguida, consideramos a implementação dos operadores usados no cálculo da iteração, pois os restantes não apresentam quaisquer dificuldades, tendo em conta o que foi exposto anteriormente sobre o multiplicador decimal.

Na implementação do multiplicador 6×7 considerou-se a configuração da figura 4.

Também neste caso, os produtos parciais que estão alinhados são somados em binário antes de serem convertidos para decimal.

Na implementação do multiplicador 10×6 considerou-se a configuração da figura 5.

Na multiplicação de 10×6 dígitos existem quatro produtos parciais que estão alinhados, são eles o três com o cinco e o dois com o quatro. Dos seis que seriam calculados só cinco o são (dois a seis) porque o resultado da operação é truncado aos primeiros dez dígitos.

O complemento foi realizado em decimal. Como tal, considerou-se o complemento para 10. O dígito B é agora negado e somado de "'1010"' para obter o complemento para nove. Posteriormente é feito o incrementado de 1 para que o valor final seja o complemento para dez.

5. Resultados de Implementação

O divisor foi descrito em VHDL e sintetizado no ambiente de projecto ISE (*Integrated Software Environment* 10.1 da Xilinx. As FPGA alvo consideradas foram a Virtex-4 SX35-12 e a Virtex-5 SX35-3.

A tabela 7 resume os resultados de implementação do divisor decimal após P&R.

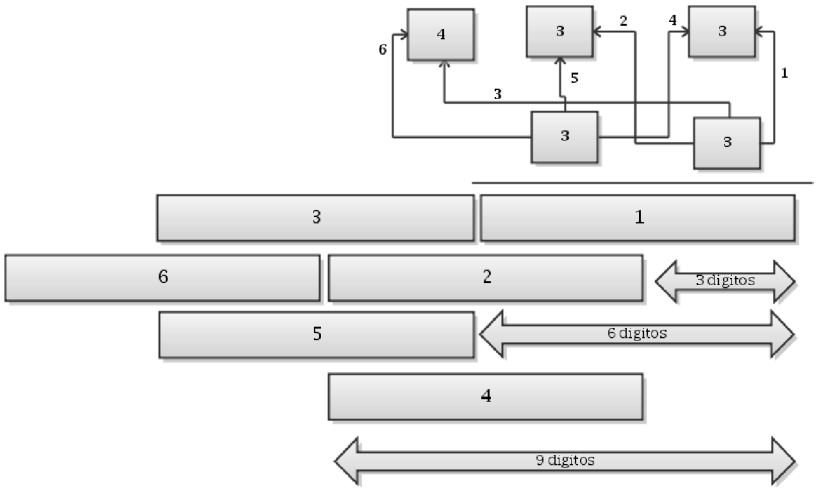


Figura 5. Multiplicação Decimal 10×6

Slices	LUT	BRAM	DSP	Freq.
1414	2642	2	11	16 MHz
1137	1917	1	11	23 MHz

Tabela 7. Resultados de implementação do divisor decimal

Os recursos utilizados pela arquitectura rondam os 9 e os 5% para as FPGA Virtex-4 e Virtex-5, respectivamente e as frequências de operação ultrapassam claramente os 10 MHz sem utilização de pipeline.

A proposta apresentada neste artigo também usa o método de Newton-Raphson, como em [6], mas com uma aproximação inicial mais eficiente, uma vez que precisa de menos uma iteração do método de Newton-Raphson.

6. Conclusões e Trabalho Futuro

O artigo apresenta a implementação de uma arquitectura de um divisor decimal numa FPGA. A solução tira partido dos multiplicadores embebidos para o cálculo das multiplicações decimais com recurso a multiplicadores binários.

Os resultados mostram que os recursos utilizados são inferiores a 10% de uma FPGA de média dimensão e as frequências de operação rondam os 20 MHz sem recurso a pipelining.

A solução apresentada será estendida para números de 16 e de 34 dígitos.

Referências

[1] *IBM Power6*, IBM Corporation, Maio 2007, http://www2.hursley.ibm.com/decimal/.

[2] M. A. Erle, M. J. Schulte e B. Hickmann, *Decimal floating-point multiplication via carry-save addition*, em Proc. IEEE Int. Conf. Application Specific Systems, Junho 2003, pp. 348-358.

[3] R. D. Kenney, M. J. Schulte e M. A. Erle, *High-frequency decimal multiplier*, em Proc. IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors, Outubro 2004, pp. 26-29.

[4] T. Lang e A. Nannarelli, *A A radix-10 combinational multiplier*, em Proc. IEEE 40th Int. Asilomar Conf. on Signals, Systems, and Computers, Outubro 2006, pp. 313-317.

[5] Mário Véstias e Horácio Neto, *Decimal Multiplier on FPGA using Embedded Binary Multipliers*, em Proc. IEEE 18th International Conference on Field Programmable Logic and Applications, Agosto 2008, pp. 197-202.

[6] L.-K. Wang e M. Schulte, *Decimal floating-point division using Newton-Raphson iteration*, em Proc. IEEE 15th International Conference on Application-Specific Systems, Setembro 2004, pp. 84-95.

[7] H. Nikmehr, B. Phillips e C.-C. Lim, *Fast decimal floating-point division*, em IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Setembro, 2006, pp. 951-961.

[8] T. Lang e A. Nannarelli, *A Radix-10 Digit-Recurrence Division Unit: Algorithm and Architecture*, em IEEE Transactions on Computers, vol. 56, nº 6, Junho 2007, pp. 727-739.

[9] A. Vazquez, E. Antelo e P. Montuschi, *A radix-10 SRT divider based on alternative BCD codings*, em Proc. IEEE 25th International Conference on Computer Design, Outubro 2007, pp. 280-287.

[10] T. Lang e A. Nannarelli, *Division Unit for Binary Integer Decimals*, em Proc. IEEE 20th International Conference on Application-specific Systems, Architectures and Processors, Setembro 2009, pp. 1-7.

Double-precision Floating-point Performance of Computational Devices: FPGAs, CPUs, and GPUs

Frederico Pratas, Aleksandar Ilic, Leonel Sousa, and Horácio Neto
INESC-ID/IST TULisbon
Rua Alves Redol, 9
1000-029 Lisboa, Portugal
{fcpp,ilic,las,hcn}@inesc-id.pt

Abstract

We have been assisting to remarkable changes in scientific computing paradigms during the last 50 years. With the increasing need for more computational power and the hit of power and memory walls, High Performance Computing has become a central discussion, pursuing for alternative solutions to increase applications performance. In fact, scientific applications have become computationally more demanding, many times requiring double-precision floating-point arithmetics. In this paper we analyze the evolution of double-precision floating-point computing for different types of devices: high-end and low-end Field Programmable Gate Arrays (FPGAs), general-purpose processors (GPPs), and graphics processing units (GPUs). We provide a per-device comprehensive survey about the performance, area, and frequency of double-precision arithmetic units during the last 9 years for the main manufacturers in the market. Our results show that peak-performance for double-precision addition and multiplication on FPGAs is already better than GPPs, and tends to keep up with GPUs.

1. Introduction

In the last few years, scientific applications have become computationally more demanding. Modern science in general and engineering in particular, have become increasingly dependent on supercomputer simulation to reduce experimentation requirements and to offer insight into microscopic phenomena. Examples of such scientific fields are Molecular and Quantum Mechanics, Bioinformatics, and Fluid Mechanics among others. Many of the applications used in these fields require *IEEE standard*, double precision floating-point operations support. In fact they require fully IEEE compliant architectures (including denormals support) to maintain numerical stability.

Such developments brought an increasing effort for the programmers/designers when deciding which type of architecture should be more efficient and/or easier to use under certain conditions. Indeed, designers have to take into account several additional parameters which are not so easy to quantify. For example, in addition to the usual time and power performance constraints they have to consider flexibility and complexity of the target architectures. Thus, depending on the application demands, different device types, namely *general-purpose processors* (GPP), *specific-purpose processors* (SPP), *reconfigurable hardware* (RH), and *Application Specific Integrated Circuits* (ASIC), must be carefully analyzed to guarantee a good balance between the application and the platform used to implement it.

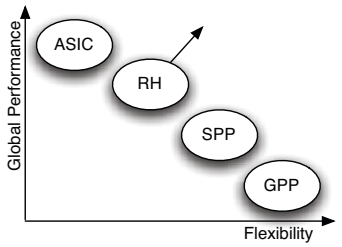


Figure 1. Global Performance vs. Flexibility

As depicted in Figure 1, GPPs have a high degree of flexibility, thus being very efficient to execute a group of different applications, but can fail to address the requirements of a specific aggressive computational application. SPPs, like *digital signal processors* (DSP) and *graphics processing units* (GPUs), can achieve better performance than GPPs for a given application at the cost of some flexibility. Both these solutions mainly use a temporal approach to implement different computational applications and are relatively easy to program. On the limit, ASICs use a spatial approach to implement only domain-specific applications. This solution is able to achieve high performance by exploiting all the application parallelism, but tends to have high non-recurring engineering and can not be reused even for similar applications. On the other hand, RH uses a combination of temporal and spacial approaches to implement multiple applications with a loose degree of similarity, i.e., the hardware is adapted to a set of applications that are loaded sequentially. The main goal of RH development is to achieve the GPP flexibility and the ASIC performance, being very efficient as a co-processing solution to accelerate certain types of applications.

While ASICs, SPPs, and GPPs devices are more mature technologies, RH is more under development. It was only in May, 1999 that the implementation of *IEEE 754* compliant, double-precision, floating-point addition and multiplication was made possible with the release of Xilinx XC4085XL. By that time *Field Programmable Gate Arrays*

(FPGAs) were in a very early stage, being outperformed by GPPs in many aspects. However, in the last decades, effects of Moore's Law have brought a dramatic impact to the semiconductor industry where the size reduction in CMOS technology allowed to double the transistors *per* unit area every two years. Consequently, processing power has also increased, not only due to the higher frequencies, but also because of the amount of processing elements implemented *per* chip. In fact, the constant growth and improvements of RH devices can be observed throughout this paper.

This evolution has been naturally supported by both the technological trends, and developments in the architectural design of FPGAs. For example in the case of Xilinx, the introduction of 18x18 multipliers into the Virtex II architecture, and later the introduction of DSP48, DSP48E, and DSP48E1 structures into the Virtex IV, Virtex V, and Virtex VI architectures, respectively, dramatically reduced the area requirements for certain implementations, and improved its efficiency. A similar evolution can be observed for Altera built-in multiplier structures. Actually, an interesting aspect that we have seen in the last few years is the evolution of FPGAs into more mixed-grained topologies. Clearly, the manufacturers try to increase the market spectrum by exploiting the best of two worlds: flexibility of the fine-grained structure, and high performance of the coarse-grained elements.

Focusing on floating-point performance, it has been increasing faster for FPGAs than for GPPs, while maintaining very low power consumptions. Indeed, it has already been shown that FPGA designs using floating-point operations can compete with GPPs in terms of peak-performance, but this study is outdated with respect to factors such as technology, number of available devices and its capacities [1]. Therefore, the aforementioned trends, coupled with the potential of FPGAs to sustain a high computational performance, prompted the peak-performance analysis performed herein considering double-precision floating-point arithmetics directly implemented in hardware. *IEEE 754* compliant double precision floating-point addition and multiplication operations are implemented on a significative set of FPGAs over the course of 9 years. In order to track the performance and area requirements we provide a comparison between FPGAs from Altera and Xilinx, as two of the main FPGA manufacturers. Trend lines are plotted according to the obtained results and are compared against known CPU and GPU data for the same time period. We also provide an analysis of technology evolution for the considered devices.

The remainder of this paper is organized as follows. Related work on floating-point operations in FPGAs is described in Section 2. Section 3 presents the implementation of the floating-point operations analyzed. Section 4 presents the obtained results for the several FPGAs and provides a comparison between FPGA and CPU performance trends. Finally, Section 5 concludes the paper.

2. Related Work

As stated in Section 1, the motivation of this work is supported by the technological development that we have been assisting in the last years. An extensive set of previous works such as [2–6] have investigated the use of custom floating-point formats in FPGAs. Some work about translation of floating-point to fixed-point format [7], and the automatic optimization of the bit widths of floating-point formats [8] has also been performed. In most cases, these formats are shown to be adequate requiring significantly less area for implementation and running significantly faster than *IEEE standard* formats [9]. However, many scientific applications, for example, from the fields of Bioinformatics such as MrBayes [10], and Molecular Chemistry such as NAMD [11], require the use of *IEEE* single- or double-precision floating-point format, not only to have more precision but also to maintain numerical stability. First works on IEEE floating-point in FPGAs, such as [12, 13], focused mostly on single-precision floating-point arithmetics and obtained relatively poor performances by that time. Later, it was demonstrated in [14] that, although being outperformed by CPUs, in terms of peak FLOPs, FPGAs were already able to provide competitive and sustained floating-point performance. Other works published in this field, namely [3, 6, 15, 16] have demonstrated the growing feasibility of *IEEE 754* compliant, or of approximately that complexity, for single-precision floating-point arithmetic and/or other floating-point formats. Actually, [17] suggests that, comparing to a GPP, a set of FPGAs can achieve a significantly higher performance. Similarly, [18] studied how to leverage new FPGA features to improve general floating-point performance. Lately, [19] studied different floating-point implementations with different precisions, and [20] provided an analysis about peak performance sustainability for three subroutines of the BLAS library. A survey comparing single- and double-precision arithmetic implementations on Xilinx FPGAs with a general-purpose CPU from 1997 until 2003 is provided in [1]. Besides, at best of our knowledge, to date only a few works focus on the performance of IEEE double-precision floating-point, and no work has provided a comprehensive performance comparison considering high-end and low-end FPGAs, GPPs, and GPUs from different manufacturers from 2001 up to today.

3. Implementation

Two floating-point *IEEE 754* compliant arithmetic operations [9] were implemented in this paper, namely double-precision addition and multiplication. Division was not considered in this case because the operations is not directly supported in GPUs hardware and a fair comparison would require to analyze how it is implemented in software for the different devices, which is not in the scope of this work. A $64-bit$ *IEEE 754* double precision format comprises three fields as depicted in Figure 2: *i*) $1-bit$ sign s; *ii*) biased exponent $e = E + 1023$; *iii*) fraction $f = \cdot b_1 b_2 ... b_{52}$. The mantissa is maintained with an implied one, i.e., it is formed by adding a "1" before the stored

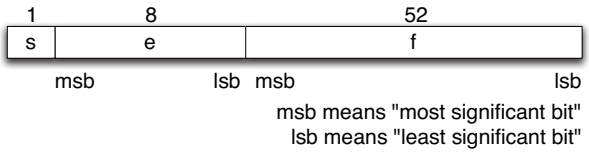
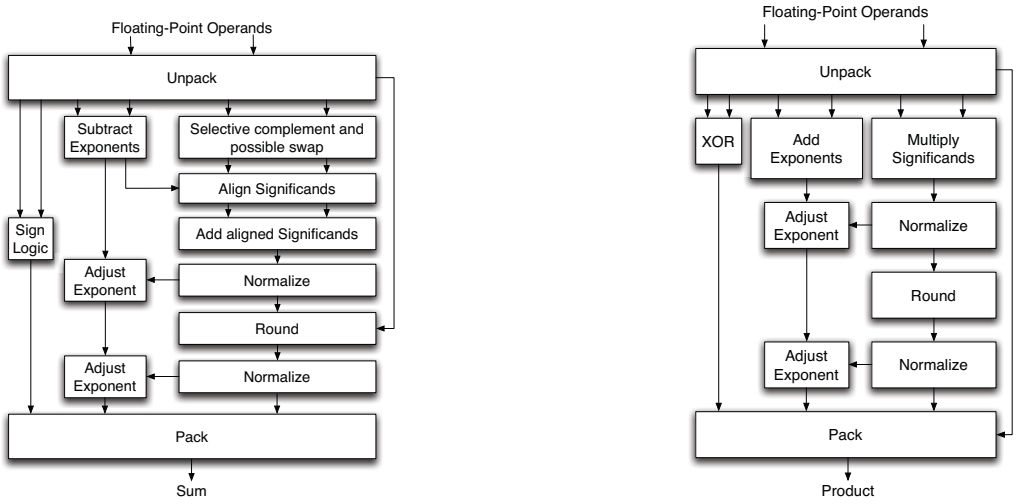


Figure 2. Double Format



(a) Adder (b) Multiplier

Figure 3. Block diagrams of floating-point operations

value, except in special cases. The decimal place is always placed immediately to the left of the stored value. Exponents of zero or the maximum field value (2047) are also reserved for special values. Thus, compliant implementations require a significant number of normalization elements. Moreover, there are a number of special values that are generated when exception conditions occur and must be handled for an *IEEE 754* compliant implementation, namely: *i*) *Not a Number* (NaN); *ii*) Infinity (∞), *iii*) Zero (0), and *iv*) denormalized numbers. The implementation should also handle these values as inputs to the operations in case of cascaded elements. Besides, gradual underflow (or denormal processing) must be provided.

In this work, the implemented floating-point units produce correct output in all exception conditions, and provide special exception signals, namely *i*) Invalid Operation, *ii*) Overflow, and *iii*) Underflow. They also provide proper handling of all special values and provide full denormal processing. Round-to-nearest-even, as defined by the *IEEE standard*, is the rounding mode provided in the implementation used herein. The discussions presented in the next sections concern the handling of exponents, alignment of significands, and normalization and rounding of results for each of the three operations considered according to [21].

3.1. Addition

A floating-point adder consists of a fixed-point adder for the aligned significands, and additional circuitry to deal with the signs, exponents, alignment pre-shift, normalization post-shift, and special values. The block diagram illustrated in Figure 3(a) shows the main components of this adder. The two floating-point operands entering the arithmetic unit are first unpacked, i.e., sign, exponent, and significand are separated, the implied "1" is reinstated, and the operands are tested for the presence of special values and exceptions. Both alignment (or pre-normalization) and post-normalization are shift operations, which provide proper handling of denormal cases with very little modification. The core floating-point operation is either an addition or subtraction (depending on the signs of the inputs). Rounding the result may require another normalizing shift and exponent adjustment. To obtain a properly rounded floating-point sum, i.e., to prevent loss of precision and correctly determine if the result should be rounded down or up, the adder must maintain at least three extra bits (*guard bit, round bit*, and *sticky bit*). Finally, packing the result involves combining the sign, exponent, and significand and removing the implied "1", as well as testing for special outcomes and exceptions (e.g., overflow, or underflow).

3.2. Multiplication

The fundamental multiplication operation is conceptually simple, as illustrated in Figure 3(b). A floating-point multiplier consists of a fixed-point multiplier for the significands, plus peripheral and support circuitry to deal with the exponents and special values (the same overall structure applies also to a floating-point divider). The role of both unpacking and packing is exactly the same as discussed for floating-point adders. The sign of the product is obtained by XORing the signs of both operands. Rounding the result may necessitate another normalizing shift and exponent adjustment as in the floating-pint adder. Multiplication only

Year	2001	2002	2003	2004	2005	2006	2007	2008	2009	Near Future
Manufacturer	**Xilinx**									
Family	Virtex 2	Virtex 2P	Virtex 2P	Virtex 4	Virtex 4	Virtex 5	Virtex 5	Virtex 5	Virtex 6	Virtex 6
Model	XC2V8000	XC2VP50	XC2VP100	XC4VLX100	XC4VLX200	XC5VLX330	XC5VLX330T XC5VSX95T [(1)]	XC5VSX240T	XC6VLX240T	XC6VLX760T XC6VSX475T [(1)]
Max. Frequency [MHz]	450	450	450	500	500	550	550	550	600	600
Process Technology [μm]	0.15/0.12	0.13/0.09	0.13/0.09	0.09	0.09	0.065	0.065	0.065	0.04	0.04
Manufacturer	**Altera**									
Family		Stratix 1	Stratix 1 GX	Stratix 2	Stratix 2	Stratix 2 GX	Stratix 3 E	Stratix 4 GX	Stratix 4 GX	Stratix 4 E
Model		EP1S80	EP1SX40G	EP2S130	EP2S180	EP2SGX130G	EP3SE110	EP4SGX230 EP4SL340 [(1)]	EP4SGX530	EP4SE820
Max. Frequency [MHz]		420	420	550	550	550	550	550	550	550
Process Technology [μm]		0.13	0.13	0.09	0.09	0.09	0.065	0.04	0.04	0.04

[(1)] Specific devices that are able to improve one of the operations, see Section 4 for more details.

Table 1. High-end FPGAs overview

Year	2003	2004	2005	2006	2007	2009	Near Future
Manufacturer	**Xilinx**						
Family	Spartan 3	Spartan 3	Spartan 3E	Spartan 3A	Spartan 3A DSP	Spartan 6	–
Model	XC3S4000	XC3S5000	XC3S1600E	XC3SD1400A	XC3SD3400A	XC6SLX150	–
Max. Frequency [MHz]	280	280	333	350	350	287	–
Process Technology [μm]	0.09	0.09	0.09	0.09	0.09	0.045	–
Manufacturer	**Altera**						
Family	Cyclone I	Cyclone I	Cyclone II	Cyclone II	Cyclone III	Cyclone III LS	Cyclone IV
Model	EP1C20	EP1C20	EP2C70	EP2C70	EP3C120	EP3CLS200	EP4CGX150
Max. Frequency [MHz]	405	405	402.5	402.5	437.5	437.5	437.5
Process Technology [μm]	0.13	0.13	0.09	0.09	0.065	0.065	0.06

Table 2. Low-end FPGAs overview

Year	2001	2002	2003	2004	2005	2006	2007	2008	2009	Near Future
Manufacturer	**Intel**									
Family	Pentium 4	Itanium II	Pentium 4	Pentium 4E	Pentium EE	Xeon				Xeon
Model	Willamette	Montvale	Northwood C	Prescott	Smithfield	Clovertown	Harpertown	Dunnington		–
Architecture Family	NetBurst	Itanium	NetBurst			Intel Core 2		Nehalem		Haswell
#Cores	1	2	1	1	2	4	4	6		8
Frequency [GHz]	2.0	1.6	3.4	3.8	3.2	3.0	3.4	2.66		4.0
Data Width [bit]	32			64						64
Power [Watt]	75	100	89	115	130	120	150	130		–
Process Technology [μm]	0.18	0.09	0.13	0.09	0.09	0.065	0.045	0.045		0.022
Manufacturer	**AMD**									
Family	Athlon XP/MP		Athlon 64		Opteron	Athlon 64 X2	Phenom X4	Opteron		Opteron
Model	Palomino	Thoroughbred	ClawHammer		Denmark	Windsor	Agena	Shanghai	Istanbul	Interlagos
Architecture Family	K7		K8				K10			Bulldozer
#Cores	1	1	1	1	2	2	4	4	6	16
Frequency [GHz]	1.73	2.25	2.6	2.6	2.8	3.2	2.6	2.9	2.8	3.0
Data Width [bit]	32		64							64
Power [Watt]	68	68	89	89	120	125	140	75	105	–
Process Technology [μm]	0.18	0.13	0.13	0.13	0.09	0.09	0.065	0.045	0.045	0.028

Table 3. GPPs overview

Year	2007	2008	2009	Near Future	2008	2009	Near Future
Manufacturer	**ATI**				**NVIDIA**		
Family	Radeon R600	Radeon R700	Radeon Evergreen	Radeon N. Islands	GeForce 200	GeForce 200	Fermi
Model	HD3850	HD4870	HD5870	–	GTX 280	GTX 285	GT 300
#Stream Cores	320	800	1600	–	240	240	512
Shader Frequency [MHz]	668	750	850	–	1236	1476	–
Power [Watt]	75	150	188	–	256	183	–
Process Technology [μm]	0.055	0.055	0.04	0.028	0.055	0.055	0.04

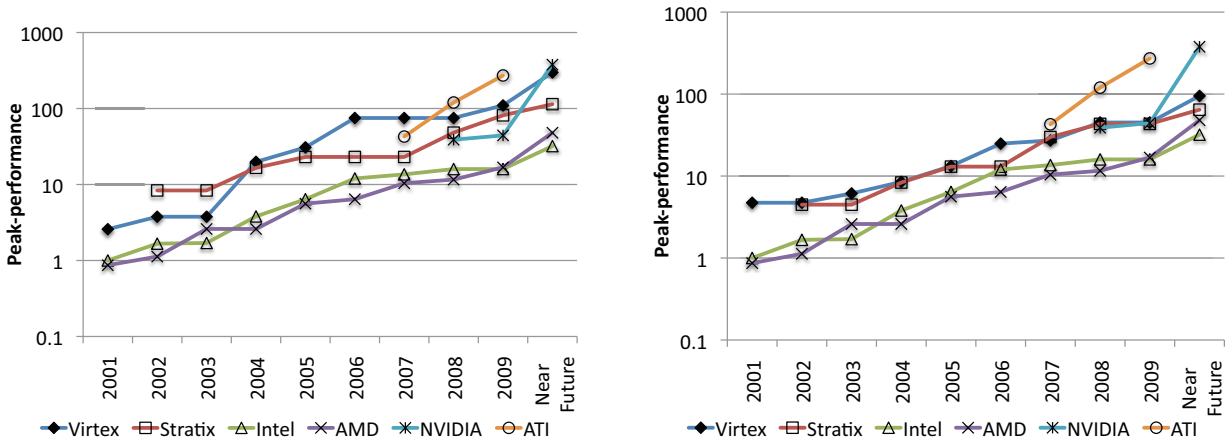
Table 4. GPUs overview

needs the *round bit* and the *sticky bit* to properly round the final result.

Compliance with the *IEEE standard* is complicated for multiplication, for example if the inputs are denormal, maintaining proper precision requires the smaller number to be normalized. If the larger input is denormal, the result will be underflow. Also, if one input is a NaN the exact value of the NaN input must be preserved and propagated. Finally, two non-denormal inputs could produce a denormal result. This requires the final output to be normalized, or rather denormalized, appropriately.

4. Experimental Setup and Results

In order to accomplish the analysis described in Section 1, a large set of experiments was performed with different FPGAs implementing double-precision floating-point addition and multiplication operations. These experiments were conducted by using both Xilinx and Altera FPGA devices released from 2001 up to today. The 27 FPGAs presented in Tables 1 and 2, are only the ones capable to deliver the highest per-operation peak-performance in each year for both manufacturers. Moreover, the FPGA results are compared with the 17 GPPs from AMD and Intel over the same 9 years, which are presented in Table 3. These processors were also selected according to their peak-performance for each year (regardless of the release date during that year). We also compare the obtained results with the highest peak-performance NVIDIA and ATI GPUs for the same period, although in this case the hardware support for double-precision floating-point arithmetics was only introduced in the devices released after 2007. Therefore we could only select the 5 best devices shown in Table 4. Additionally, in order to foresee the im-



(a) Adder Performance

(b) Multiplier Performance

Figure 4. Performance comparison with GPPs

pact of future devices, we considered models that have been announced but are not yet in the market, these are listed in the respective Tables under the column "*Near Future*".

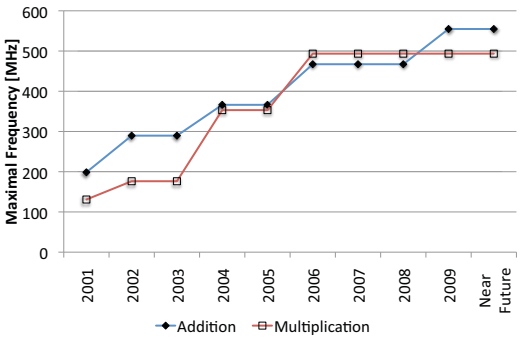
To implement the arithmetic unit in FPGA from different manufacturers we used the Xilinx ISE 11.1 and Altera Quartus II 9.1 toolkits. In particular, each arithmetic unit described in Section 3 was implemented for the Xilinx FPGAs using the *CORE Generator LogiCore Floating-Point Operator v5.0* [22] and for the Altera FPGAs using the *Floating-Point Arithmetic Functions in MegaWizard Plug-In Manager* [23]. Each design was synthesized and "Placed & Routed" for each manufacturer using two different families of boards: Virtex and Stratix as high-end FPGAs, and Spartan and Cyclone as low-end FPGAs. I/O resources were not used and constraints were adjusted to optimize final results for both speed and area with extra effort level. Furthermore, the results presented herein reflect the best case scenario when combining the implementations in logic fabrics and the specialized coarse-grained structures when applicable (i.e., hardwired DPSs or Multipliers). Moreover, the implementation of the addition units in Altera, are constrained by architectural and tool limitations which can not take advantage of the specialized coarse-grained hardware structures.

Taking into account these limitations and the different operation characteristics for certain years, more than one device is selected. This is due to the fact that some implementations are able to exploit the coarse-grain structures present in some devices, whereas the others benefit from larger amounts of fabric logic. Specifically for Xilinx in "*2007*" and "*Near Future*", the "XC5VLX330T" and "XC6VLX760T" devices are used for addition, whereas "XC5VSX95T" and "XC6VSX475T" are used for multiplication. Similarly, Altera results obtained for "*2008*" use the devices "EP4SL340" for addition, and "EP4SGX230" for multiplication.

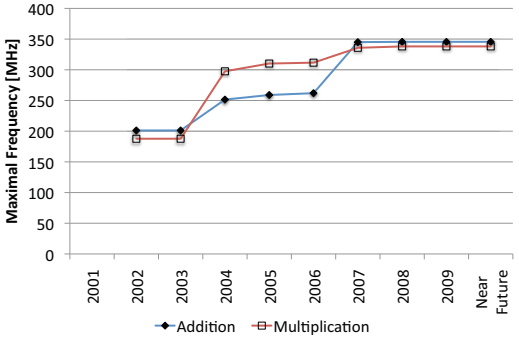
The double-precision floating-point peak-performance for each FPGA is calculated as the maximum number of functional units that can be instantiated times the worst-case operating frequency. In turn, the number of functional units that can be instantiated is simply the number of available configurable units (Slices or Logic Elements, for Xilinx and Altera, respectively) divided by the number of units required to implement one functional unit. Although these are first order approximations, this number is approximately as realistic as peak-performance values for GPPs and GPUs. Indeed, the peak-performance for each GPP and GPU is calculated as the multiplication of the processor's frequency by the number of double-precision operations that can be performed concurrently [24]. For all the devices (FPGAs, GPPs, and GPUs), peak-performance is computed individually for each considered operation (i.e., addition and multiplication).

4.1. Performance

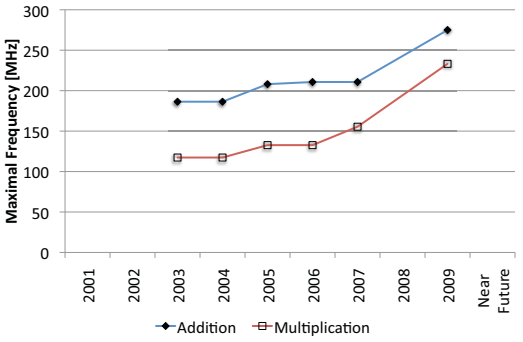
Figure 4 shows the comparison between the performance of the floating-point operations for different devices, namely GPPs, GPUs and high-end FPGAs. In particular, Figure 4(a) shows the results for the floating-point addition, and Figure 4(b) depicts the multiplication performance results. It is worth to note that the performance results are presented in logarithmic scale to facilitate the analysis of the smaller values. As expected, the results obtained for low-end FPGA devices proved their low-performance design, thus they are not presented, and they are not comparable with the high performance devices. However, the results for low-end devices are used for the analysis presented in the next Sections. As expected, all the devices show performance improvements during the considered period, in particular GPUs and FPGAs. For the addition operation FPGAs are capable of outperforming all the other devices in terms of best peak-performance, except ATI in the last two years. It is also interesting to note that since the release of Virtex 4 architecture in 2004, the Xilinx devices seem to be dominant, delivering the best performance. Also, we evidence the narrowing of a 4x performance gap between Virtex 5 and Stratix 2 in 2006, to only 1.4x in 2008 with the introduction of Altera's Stratix 4 FPGAs. Regarding multiplication, FPGAs are constantly better than GPPs,



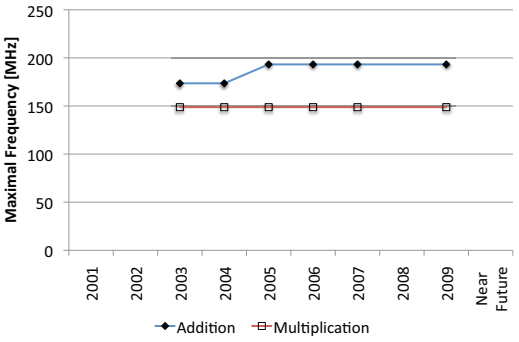
(a) Virtex



(b) Stratix

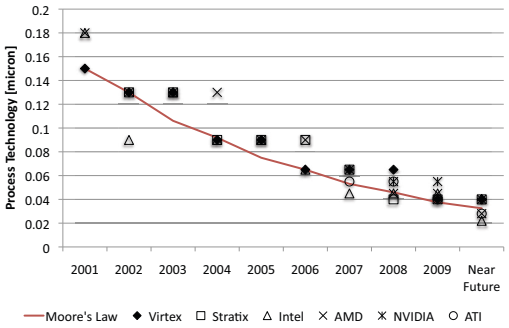


(c) Spartan



(d) Cyclone

Figure 5. Frequency Results



(a) Process Technology

Figure 6. Technology evolution

approaching the performance of NVIDIA GPUs, but still worst than ATI GPUs. All in all, nowadays GPUs clearly show the best results for both floating-point operations.

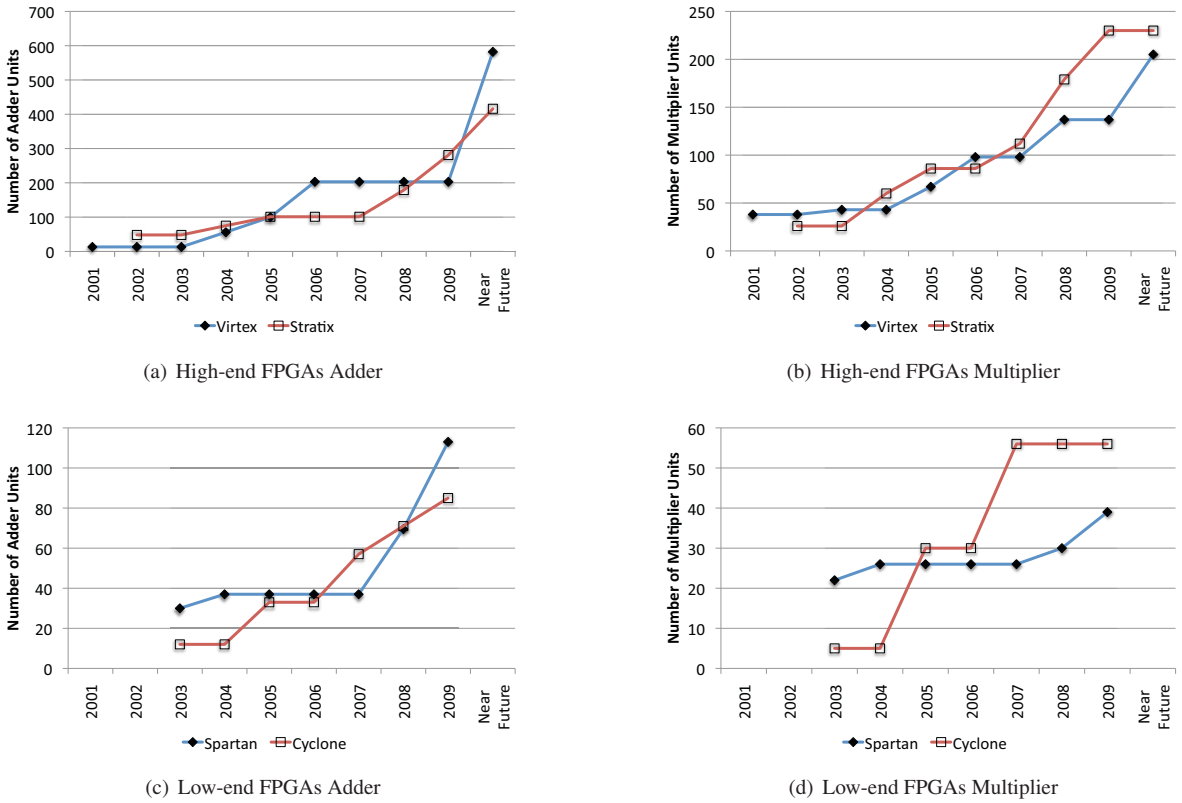
4.2. Frequency and Area

The best frequency results and maximal number of units implemented per device for the considered FPGA families are shown in Figures 5 and 7, respectively. It is worth to note that, in contrast with the previous Section, the results shown in the referred Figures are presented in different linear scales to facilitate the analysis.

Regarding the frequency, Figure 5 illustrates the results obtained with the different types of operations implemented in each FPGA family. The different complexity of the operations, creates an evident frequency offset and reduces the maximal frequency. In general the obtained frequencies follow the process technology trend, which is shown in Figure 6(a). One can observe that mainly the high-end devices tend to be affected by the trends in integration density predicted by Gordon Moore in 1965, whereas the low-end devices show stable and similar results for both manufacturers in terms of frequency.

Figure 5(a) shows that the 18x18 embedded multipliers provided in Virtex 2 and Virtex 2P are not very efficient, since the results for the multiplication operation are worst than for addition (utill 2003), which is directly implemented in the fabric. This does not stand for the embedded DSP48 family structures provided in the succeeding Virtex FPGA families. A similar pattern can be observed for the Stratix embedded devices utilization. Another aspect, common to every FGPA family (high-end and low-end) is the higher frequency obtained for the addition operation. These results are related with the usage of the highly optimized addition carry-chain structures provided in all the

978-972-789-304-1 REC'2010



(a) High-end FPGAs Adder

(b) High-end FPGAs Multiplier

(c) Low-end FPGAs Adder

(d) Low-end FPGAs Multiplier

Figure 7. Number of Units per Device

FPGA models and the DSP structures in the case of Xilinx devices. Concentrating on low-end FPGAs, both manufacturers show comparable and stable frequency results, with the exception of the newest Spartan 6 devices which obviously benefit from the new process technology.

In terms of number of units implemented per device, Figures 7(a) and 7(b) show similar results for both Virtex and Stratix high-end FPGAs until the present. Generally, significant improvements are noticeable for the years when new device families were introduced, for both Virtex and Altera. For example, for the addition operation, the release of Virtex 5 in 2006 caused a gap comparing to the Stratix 2 FPGAs available at that time, which disappeared with the introduction of Stratix 4 series. These differences are mainly related with the fact that: *i*) the *Altera MegaFunctions* tool does not use the Stratix embedded multiplier structures to implement the floating-point adders, while the *Xilinx CoreGen* tool uses the DSP48E structures; *ii*) all of the FPGA devices have drastically increased in terms of number of implemented units per device comparing to previous models. Overall, the number of units that can be implemented for any of the considered operations has increased significantly.

Regarding the low-end FPGAs one can observe from Figures 7(c) and 7(d) that until 2008 Cyclone achieved significant improvements in comparison with Spartan. However, this trend ended in 2009 with the release of Spartan 6. In general, combining the results for the maximal frequency and number of units implemented for all the devices reveals that each of these parameters can limit the acquisition of the highest overall performance.

According to these results and also to the frequency results presented previously, we can conclude that in general after 2005, with the introduction of Cyclone II, and contrarily to what happens for the high-end FPGAs, low-end Xilinx FPGAs show worst performance results than low-end Altera FPGAs. For 2009 we can again observe a change in this trend.

4.3. Future Trends

As explained before, we provide results regarding devices that have already been announced but not released, which are presented in the charts as "*Near Future*". Unfortunately we could not present results for low-end FPGAs because there are not announced releases for Spartan and it was not possible to synthesize the arithmetic units for Cyclone IV due to limitations in Quartus II toolkit. In general, the performance results shown in Figure 4 reveal the tendency of FPGAs to outperform GPPs for both arithmetic operations. Mainly for addition we can expect that they will keep up with the GPUs peak-performance or in the best case to surpass them. Regarding frequency and area presented in Figures 5 and 7, respectively, we can observe that while the frequency is expected to stabilize for the next generation of FPGAs the number of units per device is expected to increase significantly. This trend is corroborated by the fact that integration density keeps increasing but frequency becomes more stable as FPGAs approach more power demanding technologies, due also to leakage and dissipation issues.

5. Conclusions

As current technology trends continue to follow Moore's Law in delivering the benefits of higher chip density and performance along with the challenges of increased development and manufacturing complexity, the industry evidences an increasing adoption of FPGAs for next-generation system designs. In this work we provide a comprehensive survey about the performance, area, and frequency evolution of double-precision floating-point arithmetic units implemented in FPGAs devices during the last decade for major manufacturers. Moreover, we perform the fairest possible comparison with the trends of GPPs and GPUs for the same period.

Overall results shown that FPGAs are capable to deliver higher peak-performance for double-precision floating-point addition and multiplication than GPPs, and tend to keep up with GPUs. We also conclude that coarse-grain hardware structures are a good strategy to improve the implementations of the analyzed arithmetic operations both in therms of performance and area density. Although not directed to the high-performance computing market, low-end FPGA devices show an interesting evolution, mainly regarding area. As future work we plan to compare low-end FPGAs with devices used in the embedded market, e.g., DSPs.

6. Acknowledgments

We acknowledge the help of the Coreworks team to obtain some of the results presented in this work.

References

[1] K. Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 171–180. ACM New York, NY, USA, 2004.

[2] N. Shirazi, A. Walters, and P. Athanas. Quantitative Analysis of Floating point Arithmetic on FPGA based custom Computing Machines. In *FCCM '95*, page 155. IEEE CS, 1995.

[3] Pavle Belanovic and Miriam Leeser. A Library of Parameterized Floating-Point Modules and Their Use. In *FPL '02*, pages 657–666. Springer-Verlag, 2002.

[4] J. Dido et al. A Flexible Floating-point Format for Optimizing Data-paths and Operators in FPGA based DSPs. In *FPGA '02*, pages 50–55. ACM, 2002.

[5] Altaf Abdul Gaffar et al. Automating Customisation of Floating-Point Designs. In *FPL '02*, pages 523–533. Springer-Verlag, 2002.

[6] Jian Liang, Russell Tessier, and Oskar Mencer. Floating Point Unit Generation and Evaluation for FPGAs. In *FCCM '03*, page 185. IEEE Computer Society, 2003.

[7] M.P. Leong et al. Automatic Floating to Fixed Point Translation and its Application to Post-Rendering 3D Warping. In *FCCM '99*, page 240. IEEE Computer Society, 1999.

[8] Altaf Abdul Gaffar et al. Floating-point bitwidth analysis via automatic differentiation. In *FPT '02*, pages 158–165, 2002.

[9] D. Stevenson et al. An American national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, 1987.

[10] F. Ronquist and J.P. Huelsenbeck. MrBayes 3: Bayesian phylogenetic inference under mixed models. *Bioinformatics*, 19(12):1572–1574, 2003.

[11] J.C. Phillips et al. Scalable molecular dynamics with NAMD. *J. Comp. Chemistry*, 26(16):1781, 2005.

[12] B. Fagin and C. Renard. Field programmable gate arrays and floating point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(3):365–367, 1994.

[13] L. Louca, TA Cook, and WH Johnson. Implementation of IEEE single precision floating point addition andmultiplication on FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines, 1996. Proceedings*, pages 107–116, 1996.

[14] Walter B. Ligon III et al. A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs. In *FCCM '98*, page 206. IEEE Computer Society, 1998.

[15] Zhen Luo and Margaret Martonosi. Accelerating Pipelined Integer and Floating-Point Accumulations in Configurable Hardware with Delayed Addition Techniques. *IEEE Trans. Comput.*, 49(3):208–218, 2000.

[16] Xiaojun Wang and Brent E. Nelson. Tradeoffs of Designing Floating-Point Division and Square Root on Virtex FPGAs. In *FCCM '03*, page 195. IEEE Computer Society, 2003.

[17] W.D. Smith and A.R. Schnore. Towards an RCC-based accelerator for computational fluid dynamics applications. *The Journal of Supercomputing*, 30(3):239–261, 2004.

[18] Eric Roesler and Brent E. Nelson. Novel Optimizations for Hardware Floating-Point Units in a Modern FPGA Architecture. In *FPL '02*, pages 637–646. Springer-Verlag, 2002.

[19] F. De Dinechin et al. When FPGAs are better at floating-point than microprocessors. *É Normale Superieure de Lyon, Tech. Rep. ensl-00174627*, 2007.

[20] K.D. Underwood and K.S. Hemmert. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 219–228. IEEE Computer Society Washington, DC, USA, 2004.

[21] B. Parhami. *Computer arithmetic: algorithms and hardware designs*. Oxford University Press Oxford, UK, 1999.

[22] Xilinx. Floating-Point Operator v4.0. Xilinx Product Specification, April 2008.

[23] Altera. Floating-Point MegaFunctions User Guide v1.0. Altera Product Specification, March 2009.

[24] D. Strenski. FPGA Floating Point Performance–a pencil and paper evaluation. HPC Wire, January 12, 2007.

Implementação de Filtros Notch em Aritmética de Ponto Fixo

Eduardo Pinheiro[1,2], Octavian Postolache[1,3], Pedro Girão[1,2]
[1]*Instituto de Telecomunicações,* [2]*Instituto Superior Técnico,*
[3]*Instituto Politécnico de Setúbal,*
eduardo.pinheiro@lx.it.pt, opostolache@lx.it.pt, p.girao@lx.it.pt

Abstract

Diversos estudos têm sido realizados com o intuito de melhorar as implementações de filtros digitais, recorrendo a estruturas intrincadas e usando abordagens probabilísticas para analisar o comportamento do erro, embora sem cobrir os filtros notch*. Dada a recorrente implementação destes na remoção do ruído da frequência da alimentação em sinais biomédicos, o presente estudo dedica-lhes a sua atenção, avaliando a viabilidade da implementação em aritmética de ponto fixo dos filtros clássicos de resposta impulsiva infinita (IIR), Butterworth, Chebyshev e elíptico.*
Considerando os constrangimentos de implementação obtidos, é estimado o desempenho máximo da aplicação de filtros notch *estáveis em FPGA, e a influência das especificações do projecto (ordem, tipo, número de bits de entrada e dos coeficientes, factor de qualidade e frequência de corte) no comportamento do filtro, bem como as implicações e as possíveis aplicações de tais resultados.*

1. Introdução

Os filtros *notch* são muito importantes para uma ampla variedade de aplicações de instrumentação, desde as telecomunicações ao processamento de sinais biomédicos, onde comummente é necessário remover uma banda estreita ou uma única frequência do sinal medido. A implementação analógica destes filtros sofre com a deriva dos componentes e consequente instabilidade do filtro, pelo que a implementação digital é preferível, até pela facilidade de projecto de filtros de ordem e factor de qualidade elevados. Todavia, a implementação digital de filtros tem limitações na exactidão devido à precisão finita da aritmética [1–4].

Devido à facilidade com que se projectam filtros IIR digitais de elevado desempenho, a resposta do filtro é tida como assegurada, mas, particularmente quando se lida com plataformas com aritmética de ponto fixo (microcontroladores, DSP e FPGA), ou com especificações de desempenho muito exigentes, a importância da exactidão dos coeficientes do filtro multiplica-se, podendo-se falhar completamente as especificações e distorcer o sinal.

Se forem omitidos os erros do *hardware* responsável pelas conversões analógico-digital e digital-analógico, os estudos de referência que abordam este tópico, [2–9], sintetizam as fontes de erros na saída do filtro em três grupos principais :

I. Quantificação do sinal de entrada num conjunto finito de níveis discretos;
II. Representação dos coeficientes do filtro num número curto de bits;
III. Propagação de erros de arredondamento ocorridos nas operações aritméticas intermédias.

Variadas abordagens procuraram caracterizar a influência destes erros na resposta final do filtro [2–9,12–14]. Se os erros do tipo-I forem considerados com uma distribuição de probabilidade uniforme, um conjunto de ferramentas de análise está disponível para caracterizar o seu comportamento [10–14]. Erros do tipo-III são regularmente escrutinados e reduzidos, por meio do desenvolvimento de novas estruturas e variações das existentes [1–2,5,15–17], baseando-se na representação em espaço de estados e forma directa I com realimentação do erro, também referida como *noise shaping* ou *error spectrum shaping* [5,9,18].

Os erros do tipo-II também têm um conjunto importante de referências bibliográficas dedicadas à análise dos problemas que acarretam, mas sem incluir os filtros *notch*. Foram derivados alguns limites de estabilidade [6,19], uma outra abordagem explora a sensibilidade dos coeficientes [15,18–20], além da proposta de novas arquitecturas para minimizar o impacto destes erros [2,5,8,17].

Particularizando para aplicações biomédicas alguns estudos analisaram o efeito da distorção causada no sinal pelos filtros digitais [15], mas a

exequibilidade e o resultado desta implementação não foram abordados. Acrescente-se que diversos estudos da área biomédica ignoram, erradamente em certa medida, as componentes de frequência mais elevada dos sinais, implementando filtros passa-baixo ou rejeita-banda largos. Vários sinais biomédicos, como o balistocardiograma, o electrocardiograma, ou o electroretinograma, são usualmente amostrados a frequências entre os 200 Hz e os 2 kHz, com os sistemas de alta-resolução a empregarem frequentemente filtros *notch* para remover a frequência da rede.

Como os sistemas de aquisição e processamento operam a diferentes frequências de amostragem, a análise do desempenho dos filtros *notch* deve ser feita para um amplo conjunto de frequências de corte normalizadas, para que se assegure a validade do estudo para a maioria dos sinais e sistemas de processamento de sinais biomédicos. Subsequentemente, avaliam-se os requisitos para a implementação de filtros *notch* IIR em aritmética de ponto fixo, para diferentes especificações de projecto, além das modificações na resposta de um FPGA quando sujeito a estas alterações estruturais. Na secção 2 descreve-se em detalhe os procedimentos seguidos para calcular os filtros, na secção 3 é feita a análise do desempenho estimado para um FPGA da família Virtex 5, encerrando-se com as conclusões na secção 4.

2. Cálculo dos Filtros

Empregando software dedicado para o projecto de filtros, foram calculados os coeficientes usando aritmética de ponto flutuante com precisão dupla para os seguintes tipos de filtro: Butterworth, Chebyshev tipo I e II e elíptico. Considerando um vector de frequências normalizadas, Ω_0, com um total de 9 pontos por década distribuídos entre 10^{-4} e 0.3 (num total de 30 pontos), um vector de factores de qualidade, Q, também com 9 pontos por década distribuídos de 1 a 10^4 (num total 37 pontos) e filtros de ordem par de 2 a 10.

Como a quantificação dos coeficientes induz movimento dos pólos, um filtro estável ao ser quantizado pode tornar-se instável, ou, mesmo que se mantenha estável, a sua resposta pode ser inaceitável. Neste último caso, apesar de os pólos se manterem no interior do círculo unitário a quantificação é demasiado grosseira, pelo que o movimento dos pólos e dos zeros adultera de forma intolerável o comportamento do filtro. Para reduzir a deslocação dos pólos e zeros do filtro, um método valioso é a implementação do filtro em secções de 2ª ordem (decompondo um filtro de ordem N no produto de $N/2$ filtros de 2ª ordem), devido à diminuição dos valores dos coeficientes, pelo que a quantificação causará, em princípio, um menor movimento dos pólos. O impacto desta opção também será avaliado.

2.1. Definições

A frequência normalizada, Ω, define-se como o quociente entre a frequência e o ritmo de Nyquist, tendo portanto ciclos por amostra como unidade.

O factor de qualidade, Q, é o quociente entre a frequência de corte do *notch*, Ω_0, e a largura da banda rejeitada (diferença entre a frequência de corte superior e a inferior Ω_1 e Ω_2). A frequência Ω_0 é o centro da banda rejeitada, a média geométrica de Ω_1 e Ω_2. Como os resultados devem ser parametrizados em função de Ω_0 e Q, e como os algoritmos de cálculo dos filtros aceitam Ω_1 e Ω_2, empregou-se (1) para obter Ω_1 e Ω_2 das especificações em Ω_0 e Q.

$$\left\{\begin{array}{l}\Omega_0=\sqrt{\Omega_1\Omega_2}\\ Q=\dfrac{\Omega_0}{\Omega_2-\Omega_1}\end{array}\right\}\Rightarrow\left\{\begin{array}{l}\Omega_2=\dfrac{\Omega_0}{2Q}\left(1+\sqrt{1+4Q^2}\right)\\ \Omega_1=\dfrac{{\Omega_0}^2}{\Omega_2}\end{array}\right\} \quad (1)$$

Os filtros foram implementados na forma directa II, que se ilustra na Fig. 1, para o caso de um filtro de segunda ordem, (2).

$$H(z)=\frac{Y(z)}{X(z)}=\frac{b_0+b_1z^{-1}+b_2z^{-2}}{1+a_1z^{-1}+a_2z^{-2}} \quad (2)$$

A avaliação da estabilidade é feita procurando pólos da função de transferência, *H(z),* fora do círculo unitário.

O desvio do filtro de ponto fixo com n bits, face à implementação de ponto flutuante com precisão dupla (16 dígitos decimais de precisão nos cálculos, formato IEEE decimal64) [21] foi medido através do erro quadrático médio, $\varepsilon_{n\,bit}$ (em dB), comparando as amplitudes em dB da função de transferência, (3).

$$\varepsilon_{n\,bit}=\sqrt{\sum_{\Omega=\Omega_{\min}}^{\Omega_{\max}}\left[\left|H_{n\,bit}(j\Omega)\right|_{dB}-\left|H_{float}(j\Omega)\right|_{dB}\right]^2} \quad (3)$$

Ω é o vector com as frequências de teste, $H_{n\_bit}(j\Omega)$ e $H_{floatt}(j\Omega)$ as funções de transferência de ambas as implementações dos filtros.

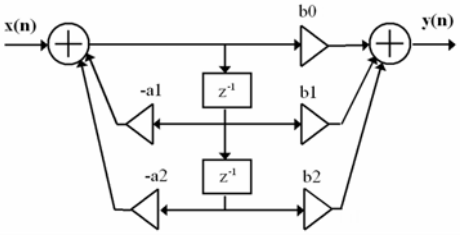


Fig. 1. Implementação de (2) na forma directa II.

Poderia ter-se definido $\varepsilon_{n\ bit}$ em unidades lineares, ou usando a diferença na fase, ou no atraso de grupo, mas como a amplitude em dB é o método mais comummente empregue para avaliar a resposta de filtros, o parâmetro $\varepsilon_{n\ bit}$ foi escolhido de forma a avaliar directamente a diferença em dB. Os desvios face à resposta ideal são problemáticos quer na banda passante quer na banda de corte, geralmente com o aumentar da exigência das especificações, manifestam-se na banda de corte e depois alastram para a banda passante, a métrica definida em (3) pesa igualmente todas as frequências.

2.2. Optimização dos filtros

A Tabela 1 apresenta o número de filtros estáveis computados, para cada ordem e para tipo de filtro, quando implementando os filtros numa única secção (SS), ou decompondo em secções de segunda ordem (SOS). A dimensão dos coeficientes considerada foi de 10 a 16 *bits*, resultando num número total de 1110 filtros para todos os pares (Q,Ω_0).

Ordem	Tipo							
	SS				SOS			
	B	C1	C2	E	B	C1	C2	E
4	134	138	162	133	1110	1110	1110	1110
6	16	19	21	16	1110	1110	1110	1110
8	6	7	6	4	1110	1110	1110	1110
10	2	3	2	1	1110	1110	1110	1105

Tabela 1. Número de filtros estáveis de ordem 4, 6, 8 e 10.

O número de bits para garantir a estabilidade de filtros decompostos em SOS varia pouco consoante o tipo de filtro. Contudo, o número máximo para garantir que todos são estáveis é bastante mais exigente que o número médio, uma vez que a grande maioria dos filtros ficam estáveis com 10 bits apenas, o que é observável na seguinte Tabela 2.

Ordem	Tipo							
	Médio (Q,Ω_0)				Máximo (Q,Ω_0)			
	B	C1	C2	E	B	C1	C2	E
4	10.11	10.13	10.10	10.13	14	14	14	14
6	10.13	10.16	10.12	10.17	14	14	14	14
8	10.14	10.17	10.15	10.20	14	15	14	16
10	10.15	10.18	10.16	---	14	15	14	>16

Tabela 2. Número de *bits* (médio e máximo) para projectar filtros estáveis para todos os pares (Q,Ω_0).

O comportamento da superfície que reflecte qual o número de *bits* óptimo é fortemente não linear, para todos os tipos e ordem de filtros, como a seguinte Fig. 2 exemplifica. Não se conseguindo obter uma relação exacta que permita, a partir das especificações de factor de qualidade e frequência de corte, saber qual o número de *bits* que se deve empregar para minimizar os desvios.

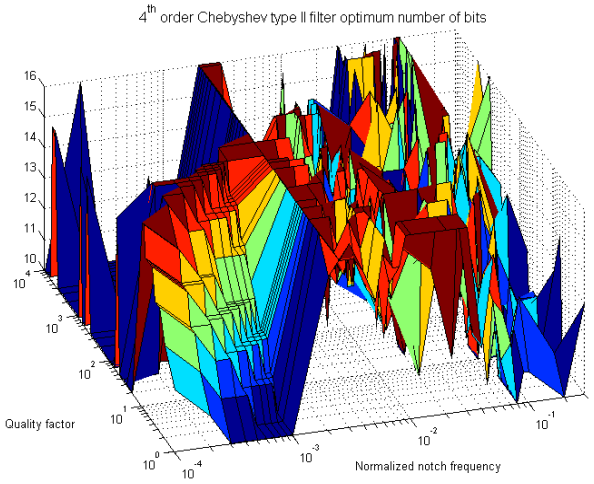


Fig. 2. Comprimento óptimo dos coeficientes para optimizar o erro, norma (3), de um filtro Chebyshev tipo I de 4ª ordem, para os vectores Q e Ω_0 definidos.

3. Desempenho do FPGA

A realização de um filtro digital na forma directa II é também conhecida como a forma canónica [22], por requerer o mínimo número de memorizações de amostras anteriores, o qual é igual à ordem do denominador da função de transferência. Uma secção de segunda ordem implementada nesta forma está ilustrada na anterior Fig. 1, correspondendo à função de transferência (2). A equação às diferenças que corresponde a este método não é a usual (4), mas sim (5).

$$y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] - a_1 y[n-1] - a_2 y[n-2] \quad (4)$$

$$\left.\begin{aligned} y[n] &= b_0 w[n] + b_1 w[n-1] + b_2 w[n-2] \\ w[n] &= x[n] - a_1 w[n-1] - a_2 w[n-2] \end{aligned}\right\} \quad (5)$$

Implementar (5) requer apenas dois registos em vez dos quatro requeridos por (4), mantendo o mesmo número de multiplicações e adições, 5 e 4 respectivamente. Observe-se ainda que todos os sinais atrasados podem ser multiplicados pelo coeficiente respectivo (a_1, a_2, b_1 e b_2) em simultâneo, como a anterior Fig. 1 ilustra, pelo que a sequência temporal da implementação consiste em duas multiplicações (a mencionada anteriormente e o produto $b_0 w(n)$) e duas adições (o cálculo intermédio de $w(n)$ e a saída final do filtro $y(n)$).

Em relação aos filtros de ordem mais elevada, apenas importa analisar os realizados em SOS,

dados os resultados anteriores. Isto irá gerar um incremento do número de operações proporcional ao incremento de ordem do filtro, por exemplo, um filtro de 4ª ordem requer 4 multiplicações e 4 adições. Dadas as características desta aplicação, o atraso devido ao percurso adicional dos dados entre secções será omitido, uma vez que não é um factor central do desempenho, pelo que irá ser estimado um majorante do desempenho óptimo, notando-se que esta latência, praticamente desprezável, irá também ser proporcional ao incremento de ordem do filtro.

3.1. Latência das operações

Dados os conversores analógico-digital (ADC) disponíveis a custo razoável, considera-se doravante que os dados de entrada no filtro têm 10, 12 ou 14 bits Os resultados que se apresentam de seguida foram obtidos com o FPGA Virtex 5 SX95T-3, estabelecendo como objectivo de optimização o desempenho temporal, sem colocar *pipelines* e para uma aritmética de ponto fixo com sinal disponibilizando a máxima precisão de saída.

As multiplicações apresentam uma subida da latência que seria perfeitamente linear se os operandos tivessem sempre um número par de *bits*, mas, como a Fig. 3 esclarece, algumas das combinações em que um ou ambos os operandos não é par, fazem a latência perder o seu comportamento linear.

Todas estas latências são bastante diminutas, devido à simplicidade da lógica envolvida, apresentando um comportamento que é aproximável por regressões lineares com erro razoável, especialmente se ambos os operandos tiverem um número par de *bits*. Os ADC comerciais possuem um número par de *bits* na saída, e, dos resultados anteriores é sabido que vários dos filtros são optimizados para um número par de *bits*, pelo que os erros de aproximação são de facto reduzidos.

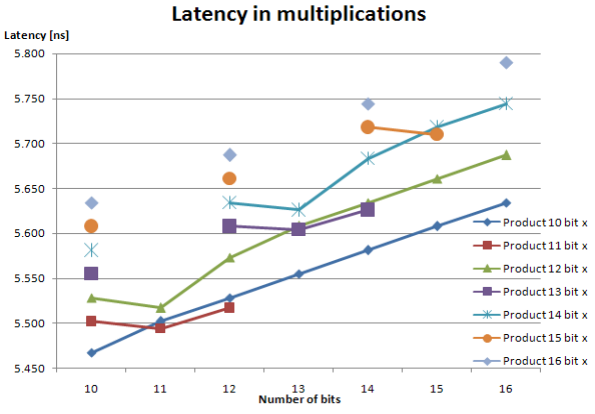


Fig. 3. Latência das multiplicações para diversas dimensões dos operandos.

A simplicidade das adições é ainda maior, variando o número de *look-up tables* linearmente com o número de operandos, com a latência a ser menos de um terço da multiplicação correspondente, ao variar entre um mínimo de 1.552 e 1.842 ns.

3.2. Latência dos filtros

Os elementos apresentados nas secções anteriores mostram que se podem recorrer a interpolações lineares para determinar a latência total do filtro com erros pouco significativos. É portanto possível majorar o melhor desempenho possível para todas as combinações de dimensão dos coeficientes e número de *bits* do ADC.

As estimativas da mínima latência de operação dos filtros são apresentadas nas seguintes tabelas 3 e 4, para um ADC de 10 *bit*. A Tabela 3 é a conversão para latência do filtro dos resultados da Tabela 2, número de bits médio e máximo dos coeficientes do filtro para garantir estabilidade para todos os pares (Q,Ω_0).

Ordem	Latência para garantir estabilidade [ns]							
	Média				Máxima			
	B	C1	C2	E	B	C1	C2	E
2	14.04	14.04	14.04	14.04	14.27	14.27	14.21	14.27
4	28.08	28.08	28.08	28.08	28.53	28.53	28.53	28.53
6	42.12	42.12	42.12	42.12	42.80	42.80	42.80	42.80
8	56.16	56.16	56.16	56.16	57.06	57.28	57.06	57.49
10	70.20	70.20	70.20	---	71.33	71.60	71.33	---

Tabela 3. Latência estimada ao implementar filtros com número de bits nos coeficientes garantindo estabilidade para todos os pares (Q,Ω_0), para um ADC de 10 *bit*.

A Tabela 4 apresenta a latência associada ao desempenho óptimo, relativamente ao erro, dos filtros.

Ordem	Latência média para minimizar o erro [ns]			
	B	C1	C2	E
2	14.11	14.16	14.11	14.16
4	28.33	28.33	28.33	28.33
6	42.34	42.48	42.34	42.48
8	56.64	56.64	56.45	56.64
10	70.81	70.81	70.80	70.81

Tabela 4. Latência estimada ao implementar filtros com número de bits garantindo erro mínimo em todos os pares (Q,Ω_0), para um ADC de 10 *bit*.

A latência para qualquer outro número de *bits* do ADC pode ser obtida por extrapolação, uma vez que se observou a linearidade do aumento da latência das

978-972-789-304-1 REC'2010

operações para um número par de operandos. As seguintes tabelas 5 e 6, apresentam a transposição para um ADC de 14 *bit* dos resultados que as duas tabelas anteriores relatam. Deve salientar-se que o produto de 14 por 13 *bits* pode ser encarado como um *outlier*, pelo que, omitindo este ponto da regressão linear, que estima a latência do produto de 14 por 11 *bits*, os resultados subsequentes são obtidos.

Ordem	Latência para garantir estabilidade [ns]							
	Média				Máxima			
	B	C1	C2	E	B	C1	C2	E
2	14.27	14.27	14.26	14.27	14.47	14.47	14.36	14.47
4	28.533	28.53	28.53	28.53	28.94	28.94	28.94	28.94
6	42.800	42.80	42.80	42.80	43.41	43.41	43.41	43.41
8	57.067	57.07	57.07	57.07	57.88	58.16	57.88	57.37
10	71.33	71.34	71.33	---	72.35	72.70	72.35	---

Tabela 5. Latência estimada ao implementar filtros com número de bits nos coeficientes garantindo estabilidade para todos os pares (Q,Ω_0), para um ADC de 14 *bit*.

Ordem	Latência média para minimizar o erro [ns]			
	B	C1	C2	E
2	14.32	14.37	14.32	14.37
4	28.74	28.74	28.74	28.74
6	42.97	43.12	42.97	43.11
8	57.49	57.46	57.30	57.49
10	71.86	71.86	71.86	71.86

Tabela 6. Latência estimada ao implementar filtros com número de bits garantindo erro mínimo em todos os pares (Q,Ω_0), para um ADC de 14 *bit*.

3.3. Discussão

Os resultados anteriores demonstraram que filtros *notch* de 2ª a 10ª ordem podem ser implementados com uma latência de algumas dezenas de nanosegundos, mesmo que com critérios de factor de qualidade e frequência de corte exigentes. Além disso, constatou-se que o número de *bits* do ADC tem uma influência pouco relevante no desempenho do FPGA, o que também acontece com a escolha do tipo de filtro. A razão para esta insensibilidade prende-se com o facto de o dispositivo da família Virtex-5 dispor de módulos DSP48E, que integram um multiplicador de 25 por 18 bits, um somador e um acumulador de 48 bits, elementos com capacidade excedente para lidar com esta gama de dimensões dos operandos.

Desta forma, após confirmar-se se o número de *bits* reservados para a quantificação dos coeficientes preservam a resposta dimensionada para o filtro, pode-se escolher o tipo de filtro e o número de *bits* do sinal de entrada, sem preocupações de desempenho, desde que o filtro seja implementado em secções de segunda ordem. Preocupações ao nível da estabilidade são residuais, uma vez que apenas quatro filtros elípticos de 10ª ordem são instáveis. A ordem do filtro revelou ser a única variável a influenciar o desempenho da implementação no FPGA.

4. Conclusões

Neste trabalho foram avaliados os efeitos das especificações de projecto, nomeadamente o factor de qualidade e a frequência de corte, no número de bits necessários para representar os coeficientes de um filtro *notch* de resposta impulsiva infinita.

Um resultado fundamental averiguado, foi a impossibilidade de aumentar a ordem do filtro acima da 2ª se este for implementar numa única secção, enquanto que, se for implementado em secções de segunda ordem a ordem pode ser aumentada até 10 sem preocupações significativas.

Constatou-se que 10 *bits* garantem a estabilidade dos filtros SOS após a quantificação dos coeficientes, para a generalidade dos casos. Usando 14 bits garante-se a estabilidade de todos os filtros elípticos, Butterworth e Chebyshev tipo I e II, até à 6ª ordem. Até à 10ª ordem 16 bits são suficientes, com excepção de 5 combinações instáveis nos filtros elípticos. A determinação do número de *bits* que minimiza o erro quadrático médio, face à implementação em ponto flutuante, deve ser feita para os valores de Q e Ω_0 que a aplicação necessite, uma vez que se constatou uma enorme irregularidade das curvas de minimização.

As estimativas de desempenho da implementação destes filtros em FPGA revelaram um processamento muito rápido, abaixo dos 80 ns num Virtex 5 SX95T-3, sendo o desempenho pouco afectado se se alterar o número de *bits* do ADC, o tipo de filtro, ou o número de *bits* dos coeficientes para a representação em ponto fixo. Como tal, a ordem do filtro é o mais destacado parâmetro a definir a velocidade de operação do dispositivo.

Em suma, os FPGA possibilitam a reconfiguração dinâmica do filtro, seja devido a alteração das especificações de funcionamento (tipo, número de bits dos coeficientes, Q e Ω_0), ou mesmo por substituição do *hardware* de entrada e saída (eventuais ADC e DAC), sem alterações notórias da frequência de operação. Esta conclusão é muito forte, uma vez que torna clara a possibilidade de implementar num único dispositivo um filtro digital dinamicamente reconfigurável com precisão variável, e capacidades de auto-adaptação para minimização de erros numéricos gerados pela

aritmética de ponto fixo, com velocidades de computação extremamente elevadas e constantes, caso a ordem não seja modificada. Adicionalmente, um filtro IIR de 2ª ordem num Virtex 5 SX95T-3 da Xilinx demora menos de 20ns a devolver o sinal filtrado. Caso a aplicação requeira uma frequência de amostragem inferior a 100 kHz, o filtro pode operar recursivamente mais de 500 vezes, erradicando completamente a frequência de corte do sinal. Este desempenho permite pensar em trabalhos futuros que implementem, com FPGA, filtros *notch* reconfiguráveis. Por exemplo no domínio da aquisição e processamento de sinais biomédicos, onde são necessários frequentemente e a sua inserção não adiciona atrasos relevantes.

Apesar de ter sido introduzido um conjunto de considerações práticas e aproximações no cenário conceptual que se desenvolveu, o estudo abre caminho a novos desenvolvidos usando FPGA no processamento digital de sinal, confirmando estes dispositivos como uma solução muito poderosa neste campo.

Agradecimentos

Pelo apoio dado a este trabalho, ao professor Horácio Neto e ao investigador Frederico Pratas, ao Instituto de Telecomunicações e à Fundação para a Ciência e Tecnologia, bolsa SFRH/BD/46772/2008.

Referências

[1] H. Cheng e G. Chiu, "Finite Precision Controller Implementation – Limitation on Sample Rate", in *Proc. IEEE/ASME International Conference on AIM 2003*, pp. 634-639, Kobe, Japan, 2003.

[2] V. Davídek, M Antosová e B. Psenicka, "Finite Word-Length Effects in Digital State-Space Filters", *Radioengineering*, vol. 8, nº. 4, pp. 7-10, December 1999.

[3] B. Liu, "Effect of Finite Word Length on the Accuracy of Digital Filters – a Review", *IEEE Transactions on Circuit Theory*, vol. CT-18, nº. 6, pp. 670-677, November 1971.

[4] H. Butterweck, J. Ritzerfeld e M. Werter, "Finite Wordlength Effects in Digital Filters: A Review", Fac. Of Elec. Eng., Eindhoven University of Technology, Netherlands, EUT Report 88-E-205, 1988.

[5] T. Laakso, J. Ranta e S. Ovaska, "Design and Implementation of Efficient IIR Notch Filters with Quantization Error Feedback", *IEEE Transactions on Instrumentation and Measurement*, vol. 43, nº. 3, pp. 449-456, June 1994.

[6] R. Otnes e L. McNamee, "Instability Thresholds in Digital Filters Due to Coefficient Rounding", *IEEE Transactions on Audio and Electroacoustics*, vol. AU-18, nº. 4, pp. 456-463, December 1970.

[7] J. Datorro, "The Implementation of Recursive Digital Filters for High-Fidelity Audio", *Journal Audio Engineering Society*, vol. 36, pp. 851-878, November 1988.

[8] M. Er, "Designing Notch Filters with Controlled Null Width", *Signal Processing*, vol. 24, pp. 319-329, September 1991.

[9] T. Laakso e I. Hartimo, "Noise Reduction in Recursive Digital Filters Using Higher-order Error Feedback", *IEEE Transactions on Signal Processing*, vol. 40, pp. 1096-1107, May 1992.

[10] J. Wilkinson, *Rounding Errors in Algebraic Processes*, Englewood Cliffs, New Jersey, 1963.

[11] G. Forsythe e C. Moller, *Computer Solution of Linear Algebraic Systems*, Englewood Cliffs, New Jersey, 1967.

[12] B. Widrow, "Statistical analysis of amplitude quantized sampled-data systems", *AIEE Transactions Appl. Ind.*, vol. 79, pp. 555-568, January 1961.

[13] J. Knatzenelson, "On errors introduced by combined sampling and quantization", *IRE Transactions on Automatic Control*, vol. AC-7, pp. 58-68, April 1962.

[14] J. Knowles e R. Edwards, "Effects of a finite-word-length computer in a sampled-data feedback system", *IEE Proceedings*, vol. 112, pp. 1197-1207, June 1965.

[15] C. Weaver, J. Van der Groeben, P. Mantey, J. Toole, C. Cole Jr., J. Fitzgerald e R. Lawrence, "Digital Filtering with Applications to Electrocardiogram Processing", *IEEE Transactions on Audio and Acoustics*, vol. AU-16, nº. 3, pp.350-391, September 1968.

[16] K. Liu, R. Skelton e K. Grigoriadis, "Optimal Controllers for Finite Wordlength Implementation", *IEEE Transactions on Automatic Control*, vol. 37, nº. 9, pp. 1294-1304, September 1992.

[17] G. Yan, "New Digital Notch Filter Structures with Low Coefficient Sensitivity", *IEEE Transactions on Circuits and Systems*, vol. CAS-31, nº. 9, pp. 825-828, September 1984.

[18] R. Goodal, "A Practical Method for Determining Coefficient Word Length in Digital Filters", *IEEE Transactions on Signal Processing*, vol. 40, nº. 4, pp. 981-985, April 1992.

[19] J. F. Kaiser, "Digital filters", in System Analysis by Digital Computers, F. F. Kuo and J. F. Kaiser, Eds. New York: Wiley, 1966.

[20] P. Mantey, "Eigenvalue sensitivity and state-variable selection", *IEEE Transactions on Automatic Control*, vol. AC-13, nº. 3, pp. 263-269, June 1968.

[21] IEEE, IEEE 754-2008 Standard for Floating-Point Arithmetic, August 2008.

[22] A. V. Oppenheim e R. W. Schafer, *Digital Signal Processing*, Englewood Cliffs, New Jersey, 1975.

Sessão Regular 5

Instrumentação e Controlo

Moderação: José Augusto
Fac. de Ciências da Univ. de Lisboa / INESC-ID

Instrumento de Análise e Diagnóstico em Máquinas Rotativas de Indução Baseado em FPGA

Cesar da Costa, Mauro Hugo Mathias
Faculdade de Engenharia - Departamento de Mecânica
UNESP-Universidade Estadual Paulista Julio de Mesquita Filho
Guaratingueta, São Paulo, Brasil
e-mail: cost036@attglobal.net, mathias@feg.unesp.br

Pedro Ramos, Pedro Silva Girão
Departamento de Engenharia Electronica e Computadores
Instituto de Telecomunicações, Instituto Superior Tecnico, Universidade Técnica de Lisboa
Lisboa, Portugal
e-mail: pedro.ramos@lx.it.pt, psgirao@ist.utl.pt

RESUMO

Atualmente o m*onitoramento das condições de operação de máquinas rotativas de indução é utilizado para aumentar a disponibilidade e o desempenho desse tipo de máquina, reduzindo os conseqüentes danos, aumentando a vida útil da máquina, reduzindo os estoques de peças sobressalentes e reduzindo a sua manutenção. Neste trabalho, uma nova metodologia para desenvolvimento de projeto de um instrumento de análise e diagnóstico, em tempo real, para máquinas rotativas baseado em arquitetura reconfigurável, mapeada de um modelo criado no software MATLAB/Simulink é apresentada. Neste estudo, as aplicações de processamento de sinal, como filtros FIR e transformada rápida de Fourier são tratadas como sistemas independentes, que são implementados em hardware embutido (FPGA) usando uma ferramenta de software, denominada DSP Builder, que traduz automaticamente um modelo desenvolvido em MATLAB/Simulink em uma linguagem de descrição de hardware – VHDL, para configuração direta em FPGA.*

I. INTRODUÇÃO

A análise de vibração em máquinas rotativas de indução é uma das ferramentas mais importantes para a identificação de falhas do equipamento. Na verdade, os grandes sistemas electro mecânicos muitas vezes são equipados com sensores baseados em quantidades mecânicas. Em muitas situações, os métodos de monitoramento de vibrações são usados para detectar a presença de falhas incipientes. No entanto, tem sido sugerido que o monitoramento da corrente do estator pode fornecer as mesmas indicações, sem exigir acesso ao motor [1]. Para efeitos de detecção de sinais relacionados com falhas, muitos métodos de diagnóstico têm sido desenvolvidos até o presente momento. Esses métodos para identificar falhas em máquinas rotativas podem envolver diversos tipos de campos da ciência e tecnologia [2,3 e 4].

II. FALHAS EM MÁQUINAS ROTATIVAS

A máquina rotativa é considerada uma máquina robusta e tolerante a falhas. O motor de indução AC é uma máquina rotativa elétrica destinada a operar a partir de uma fonte trifásica de tensão alternada. É importante que as medições de vibração e corrente sejam tomadas para diagnosticar o estado da máquina, antes dela entrar em um modo de falha. Além disso, é necessário fazê-lo em tempo real, monitorando continuamente as variáveis da máquina. Os motores de indução à semelhança de outras máquinas eléctricas rotativas são submetidos a duas forças: electromagnética e mecânica. A concepção de um motor é tal que a interação entre essas forças sob condição normal leva a um funcionamento estável, com o mínimo de ruído e vibração. Quando uma falha ocorre, o equilíbrio entre essas forças se perde, reforçando ainda mais a falha. As falhas de motores de indução podem ser classificadas em dois tipos: mecânica e eléctrica [5].

III. IDENTIFICAÇÃO DE FALHAS MECÂNICAS

A medição de vibração em tempo real, processamento e análise do sinal são ferramentas importantes para a identificação de falhas mecânicas. Existem dois tipos de análise: no domínio do tempo e domínio da freqüência. A análise no domínio da freqüência é mais atraente porque pode dar informações mais detalhadas sôbre o diagnóstico (estado) da máquina. A análise no domínio do tempo pode fornecer informações qualitativas sôbre a condição da máquina. Geralmente a vibração da máquina é um sinal estacionário composto de vibrações e ruídos. Tradicionalmente, a FFT (Fast Fourier Transform) é utilizada para fazer a análise no

domínio da frequência. Se o nível de vibração e ruído é elevado, informações imprecisas sobre a condição da máquina é obtida. O ruído e a vibração podem ser separados a partir do sinal de vibração usando filtros FIR [6].

IV. Modelagem de um Instrumento de Analise e Diagnóstico em Máquinas Rotativas

Neste estudo de caso, um instrumento digital de medição e análise de vibrações para identificação de falhas e diagnóstico em máquinas rotativas está sendo desenvolvido. No primeiro estágio do desenvolvimento, um conjunto de algoritmos de medição e análise foi desenvolvido em MATLAB/Simulink. No segundo estágio, os algoritmos de medição e análise são modelados, simulados e convertidos para código VHDL automaticamente, utilizando uma ferramenta de software denominada DSP Builder, da empresa Altera, sem a tradicional interrupção do fluxo de projeto, para codificação manual em uma linguagem de descrição de hardware – VHDL e posterior configuração da FPGA.

O software Simulink é um sistema dinâmico e de simulação interativa, que oferece um ambiente científico e de engenharia para a modelagem de sistemas, análise e simulação. Este ambiente é útil para uma rápida implementação de uma aplicação de processamento digital, em termos de blocos funcionais, e proporciona uma simulação de alto nível. Um bloco funcional é uma estrutura básica que pode representar uma função, ou um algoritmo especializado, que define portas de entrada e saída e parâmetros personalizados.

Neste trabalho, a implementação do instrumento de análise e medição de vibrações com diagramas de blocos funcionais, permitiu a modelagem dos algoritmos de medição e análise, o estudo do fluxo de dados e os testes de desempenho do sistema. O Simulink utiliza um conjunto de bibliotecas para representar o comportamento dinâmico do instrumento [8].

A. Blocos Funcionais

A modelagem dos blocos funcionais do instrumento é apresentada a seguir:

1) Modelo de aquisição de dados e filtros: o sinal proveniente do sensor de vibração e/ou corrente chega a placa de aquisição de dados, que é um cartão de 16-bit com uma taxa máxima de amostragem de 1,25 MS/s. O modelo criado no Simulink consiste em um bloco de entrada analógica para acquisição do sinal do sensor, um bloco de escalonamento linear do sinal, um bloco de filtro passa-baixa Butterworth de sexta ordem, com uma frequência de corte de 12 kHz, e um bloco de banco de quatro filtros passa-alta. Os filtros são utilizados para duas finalidades: (i) para atenuar o ruído e componentes de freqüência indesejados e (ii) para separar algumas freqüências individuais ou banda de frequências para análise de sua relação com as falhas da máquina [5]. A saída é um gráfico no domínio do tempo, da amplitude do sinal de vibração da máquina em função do tempo. Um bloco funcional que representa o modelo da aquisição de dados e os filtros é mostrado na Figura 1.

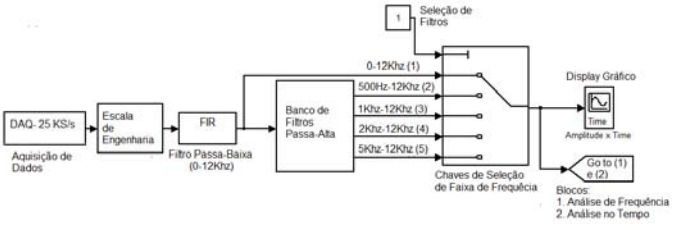


Figura 1. Modelo da aquisição de dados e os filtros de entrada do instrumento.

2) Modelo de análise no domínio do tempo: a abordagem mais simples no domínio do tempo é a medida do nível RMS (Root-Mean-Square), valor de pico, fator de crista (relação entre valor de pico e o valor RMS) do sinal de vibração. Alguns parâmetros estatísticos, tais como densidade de probabilidade e curtose foi proposto para detecção de falhas mecânicas de rolamento. A densidade de probabilidade da vibração de um rolamento em bom estado tem uma distribuição de Gauss, ao passo que um rolamento danificado tem como resultado uma distribuição não Gausiana com caudas dominantes por causa de um aumento relativo do número de níveis elevados (picos) de vibração. Em vez de estudar as curvas de densidade de probabilidade, muitas vezes é mais informativo examinar os momentos estatísticos dos dados. O primeiro e segundo momentos são bem conhecidos, sendo o valor médio e a variância, respectivamente. O terceiro momento normalizado com relação ao cubo do desvio padrão é conhecido como o coeficiente de assimetria. O quarto momento, normalizado em relação à quarta potência do desvio padrão é chamado curtose e é bastante útil. O modelo consiste de um bloco de cálculos e processamento de variáveis, e um bloco de chaves de seleção da variável desejada. A análise de sinais no domínio do tempo é utilizada para extrair algumas características úteis do sinal, ou seja, o valor médio, média quadrática, valor RMS, fator de crista e curtose. A saída mostra o valor global numérico, no domínio do tempo, da variável selecionada pelo bloco de chaves. Um bloco funcional que representa o modelo de análise no domínio do tempo é mostrado na Figura 2.

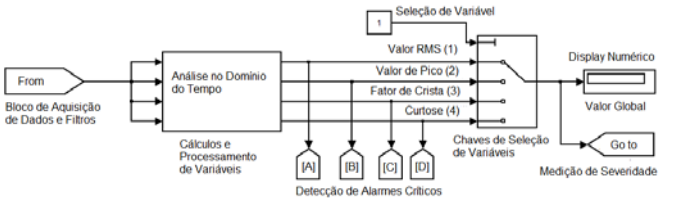


Figura 2. Modelo de análise no domínio do tempo do instrumento.

3) Modelo de medida de severidade de vibração: algumas falhas podem ser detectadas na análise do domínio do tempo e exibidas num alarme, devido à intensidade do nível de vibração global RMS especificado pela norma ISO 10816-1 (ex-ISO 2372). Um procedimento comum de controlo de falhas em máquinas rotativas é a medida do valor RMS da velocidade de vibração, chamada severidade da vibração, que é uma medida da energia das vibrações totais emitidas. O modelo consiste de um bloco de detecção da severidade de

978-972-789-304-1 REC'2010

vibração e quatro saídas de indicação do estado da máquina: boa, satisfatória, insatisfatória e inaceitável. Para detecção da severidade é utilizada os valores de intensidade de vibração especificado pela Norma ISO 10816-1, classe I, máquinas de pequeno porte.

4) Modelo de detecção de alarmes críticos: valores altos de curtose revelam grandes picos do sinal de vibração. Falhas diferentes em rolamentos produzem altos picos de vibrações, que têm componentes de alta freqüência. O valor RMS, juntamente com o fator de curtose, valor de pico e fator de crista do sinal de vibração são bons indicadores para distinguir entre um rolamento saudável e um com falha, mas nenhum deles permite fazer um diagnóstico do tipo de falha. Para um rolamento danificado com a distribuição de Gauss, o valor de curtose é próximo de 3. Um valor maior que 3 é crítico e considerado por si só uma indicação de falha iminente, sem histórico prévio. Da mesma forma, valor de pico em séries temporais irá resultar em um aumento no valor do factor de crista. Para operações normais, o valor de pico pode atingir entre 3,8 e 15 mm/s e factor de crista entre 2 e 6. Um valor elevado de pico, acima de 15 mm/s e um factor de crista acima de 6 são críticos e associados a problemas de máquinas [5]. O modelo consiste de um bloco de detecção de alarmes críticos e de saídas para exibição de três status de alarmes. As saídas de alarmes são exibidas quando o valor de pico é maior do que 15 mm/s, o factor de pico é maior do que 6, e a curtose é maior do que 3.

5) Modelo de análise no domínio da frequência: domínio da freqüência ou análise espectral do sinal de vibração é talvez a abordagem mais amplamente utilizado para identificação de falhas de máquinas rotativas. O advento da moderna análise por meio da transformada rápida de Fourier (FFT) tem feito o trabalho de obtenção de espectros de banda de frequências, mais fácil e eficiente [5]. Ambas as faixas alta e baixa do espectro de frequência de vibração são de interesse para diagnosticar a condição da máquina. A faixa de medição da freqüência é usada para obter uma visão preliminar sobre o estado da máquina. Faixas de freqüência no espectro de vibração são selecionadas de acordo com a origem da falha. O valor RMS dessas bandas é usado para especificar o grau e a origem das falhas, por comparação com as bandas correspondentes no espectro de referência de uma máquina sem falhas. A partir da literatura e observações experimentais, quatro bandas de frequências são selecionadas para cobrir os harmônicos de vibração de origem mecânica e eletromagnética [5]. A Figura 3 apresenta o modelo, que consiste de um bloco de filtros passa banda, um bloco de chaves de comutação e um bloco de cálculo da FFT. A saída produz um gráfico de amplitude versus freqüência da banda de freqüência do sinal de vibração selecionado, pelo bloco de chaves de comutação.

V. RESULTADOS EXPERIMENTAIS

Esta seção apresenta as fases iniciais do desenvolvimento do instrumento de medição e análise de vibrações, que tem sido focado no desenvolvimento e testes de algoritmos e métodos para a detecção, em tempo real, de falhas em motor de indução AC.

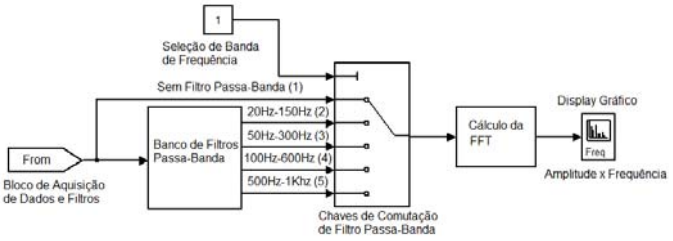


Figura 3. Modelo de análise no domínio da freqüência do instrumento

Os resultados preliminares experimentais foram obtidos a partir de um motor de indução trifásico, tipo "slip-ring", sensor de vibração, sensor de corrente, placa de aquisição de dados e um Kit de desenvolvimento da Altera, baseado na FPGA da família Cyclone II.

A carga do motor é fornecida por meio de um gerador de frenagem DC. As correntes do estator e as vibrações são adquiridas através de sensores específicos e uma placa de aquisição de dados. O sinal adquirido é processado no Kit de desenvolvimento, que contém a aplicação do instrumento de análise de diagnóstico embutida na FPGA. A Fig. 4 apresenta o motor de indução AC, na plataforma de testes montada no laboratório do Instituto de Telecomunicações do IST.

Figura 4. Motor trifásico de indução slip-ring no laboratório do IST.

A. Sensor de Vibração

O sensor de vibração é um acelerômetro, tipo MEMS, com dois eixos X e Y, faixa de +/- 2 g, sensibilidade de 420 mV/g, e largura de banda de 0,5 a 5,5 kHz para todos os eixos, fabricado pela Analog Devices (ADXL322JCP). Este dispositivo utiliza uma técnica de medição capacitiva. A estrutura mecânica do transdutor é apoiada por um circuito electrónico, que converte a alteração da capacitância devido à vibração, em uma tensão.

B. Sensor de Corrente

O sensor de corrente é baseado em um transdutor de corrente (LEM LA 25 - NP), largura de banda 0 a 100 kHz, com 0,2% de precisão, linearidade < 0,1%.

C. *Placa de Acquisição de Dados*

A placa de aquisição de dados utilizada é a DAQ, NI USB-6251, com 16-bits, 1,25 MS/s da National Instruments.

D. *Kit de Desenvolvimento*

O Kit de desenvolvimento utilizado é o Cyclone II FPGA Starter Development Board, baseado na FPGA Cyclone II EP2C20F484C7N, com 8 Mbyte de SDRAM, 512 Kbyte de SRAM e 4 Mbyte de Flash, equipado com a placa Highspeed A/D e D/A Daughter Board (ADA-GPIO) da empresa Terasic, para conversão do sinal analógico em digital, com dois canais A/D de 14 bits de resolução e taxa de 65Ms/s de amostragem.

Numerosos testes com o motor de indução estão sendo realizados no laboratório em diferentes níveis de carga. Simulação de falhas do motor e comportamental do instrumento proposto foram realizados, usando os softwares MATLAB/Simulink, DSP Builder para gerar automaticamente o código VHDL e Quartus II para testar e configurar a FPGA no Kit de desenvolvimento. A frequência de amostragem nos experimentos apresentados neste trabalho são de 2 KS/s para a análise de corrente e 25 KS/s para análise de vibração. A Fig. 5 mostra um espectro típico do motor com simulação de falhas de barras quebradas. Pode ser visto que os componentes de freqüência desejados podem ser distinguidos facilmente. Para cada tipo de espectro, há diferentes algoritmos para a extração das características importantes para o diagnóstico.

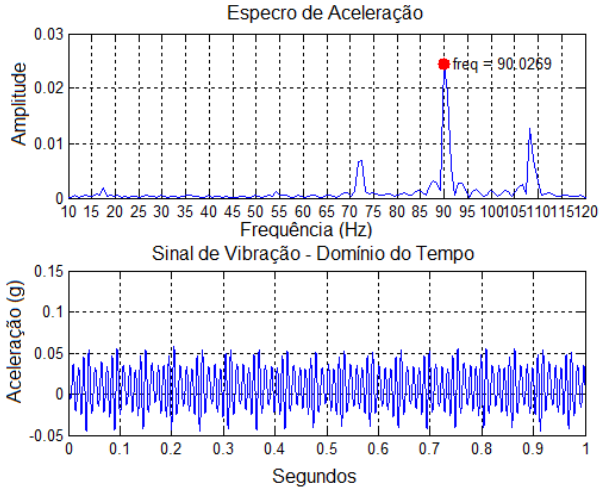


Figura 5.Espectro de vibração do motor de indução com falha de barras quebradas.

Os resultados obtidos experimentalmente com o instrumento são comparados com o modelo de simulação Simulink. Depois que o modelo gráfico é verificado e testado sem erro, o software DSP Builder é utilizado para transformar o modelo gráfico automaticamente em linguagem VHDL e o software Quartus II, simula, testa e configura a FPGA no Kit de desenvolvimento. O software DSP Builder é uma ferramenta para o desenvolvimento de processamento digital de sinais, criado pela empresa Altera, que gera automaticamente o código em linguagem de descrição de hardware (VHDL), a partir de um modelo gráfico desenvolvido em Simulink, faz a simulação em nível de algoritmo, e configura o projeto de hardware na FPGA [8].

VI. CONCLUSÃO

Uma nova abordagem para a modelagem, simulação e projeto de um instrumento embutido de análise e diagnóstico de falhas, em tempo real, baseado em lógica reconfigurável (FPGA) foi apresentado. A lógica reconfigurável possibilita um melhor desempenho que os processadores DSPs tradicionais. Entretanto, programar manualmente em VHDL algoritmos de processamento digital de sinais em uma FPGA pode ser uma tarefa longa e difícil. Até recentemente, os algoritmos necessários para ser transportado para VHDL e então fazer a simulação funcional RTL, utilizavam vetores de alto nível de testes de simulação. Ferramentas de projeto modernas como o software DSP Builder proporcionam um maior nível de abstração de projeto e de produtividade, e proporcionam o desempenho otimizado da programação tradicional VHDL. Esta abordagem utiliza uma descrição de alto nível do comportamento do algoritmo funcional. O projeto pode ser simulado e implementado em hardware na FPGA. Não é necessário usar VHDL para a programação. É apenas necessário construir um modelo do sistema desejado em Matlab/Simulink, e então o projeto pode ser concluído depois que os parâmetros de cada modelo são definidos.

REFERENCIAS

[1] M. E. H. Benbouzid, “A review of induction motors signature analysis as a medium for faults detection ”, IEEE Transactions on Industrial Electronics, Vol. 47, no. 5, pp. 984-993, Oct. 2000.

[2] A. Sadoughi, M. Ebrahimi, M. Moalem and S. Sadri,” Intelligent diagnosis of broken bars in induction motors based on new features in vibration spectrum”. Diagnostic for Electric Machines, Power Electronics and Drives, IEEE International Symposium, pp. 106-111, Sept. 2007.

[3] S. Nandi, H. A. Toliyat and X. Li, “Condition monitoring and fault diagnosis of electrical motors – a review”, IEEE Transactions on Energy Conversion, vol. 20, no. 4, pp. 719-729, Dec. 2005.

[4] J. J. Rangel- Magdaleno, R. J. Romero- Troncoso, L. M. Contreras-Medina And A. Garcia-Perez. “FPGA implementation of a novel Algorithm for on-line bar breakage detection on induction motors”. IEEE International and Measurement Technology Conference, IIMTC 2008, pp. 720-725, Canada, 2008.

[5] S. A. S. Al Kazzas and G.K. Singh, K, “Experimental investigations on induction machine condition monitoring and fault diagnosis using digital signal processing techniques”. Electric Power Systems Research, no. 65, pp. 197-221. New York: Elsevier, 2003.

[6] A. Lebaroud and G. Clerc, “Diagnosis of induction motor faults using instantaneous frequency signature analysis”. IEEE Proceedings of the 2008 International Conference on Electrical Machines, ICEM 2008, pp. 1-5, Sept. 2008.

[7] I. Grout, J. Ryan and T. O´Shea, “Configuration and debug of field programmable gate arrays using Matlab/Simulink”, Journal of Physics: Conference Series 15, pp. 244-249, 2000.

[8] G. Xiong, X. Zhou and P. Ji. “Implementation of the Quadrature Waveform Generator Based on DSP Builder”. IEEE Computer Society, International Symposium on Intelligent Information Technology Application Workshops, IITAW´08, pp. 773-776, China, Dec. 2008.

The Performance Impact when Optimizing Mapping Algorithms for an FPGA-based Mobile Robot

Manuel Luís C. Reis
FEUP
mreis@fe.up.pt

João M. P. Cardoso
FEUP
jmpc@fe.up.pt

João P. C. C. Ferreira
INESC Porto, FEUP
jcf@fe.up.pt

Abstract

FPGA-based solutions are being used to meet performance requirements in embedded systems. The mobile robotics field is an appropriate domain to evaluate whether FPGA-based systems are capable of managing complex computing tasks. This paper presents an autonomous mobile robot prototype that includes an FPGA board as the main central processing component.

In this work we evaluate the prototype using mobile robotics mapping algorithms (i.e., algorithms able to build a map of the environment). These algorithms rely mainly on probabilities and are computationally intensive. The implementation used in this paper is based on an environment occupation grid. An uncertainty model of the sensor used to measure the distance of the robot to the obstacles in the environment is used to update the probabilities.

We start the experiments by analysing the performance impact of using hardware modules, such as FPU and cache memory. Then, we consider modifications in the updating algorithm to reduce the overall execution time. The overall improvements allowed for an execution time of the updating task of 9.83 ms for an implementation based on Bayes's algorithm and 10.86ms for Dempster-Shafer's algorithm, both at the maximum distance considered.

1. Introduction

Field Programmable Gate Arrays (FPGAs) are recognised as a solution for developing embedded systems that provides the performance of dedicated hardware systems and the flexibility of reconfigurable computing [1,2].

Despite research addressing this topic being quite recent, some interesting projects involving FPGA-based controllers for embedded applications have been reported. PROTOS [3] presents a mixed solution of microcontroller and logic gates directed at high number of outputs and low access memory requirements applications. In [4] a more recent work uses a mobile robot with laser range finder sensors for mapping. Another interesting project addresses FPGA dynamic reconfiguration to match different control strategies in a mobile robot [5].

The purpose of our work is to build a prototype of an autonomous mobile robot to implement and study embedded system architectures based on FPGAs. Specifically, the main objectives for this work are:

a) to build a prototype robot that can be controlled by algorithms implemented on FPGAs;
b) to implement and study the performance of a representative mobile robotics mapping algorithm;
c) to evaluate the impact of program modifications on the overall execution time for a system with a RISC processor embedded in the FPGA.

The mobile robot prototype is based on a Vex robotic kit [6] and an FPGA board coupled to the kit. The prototype is shown in Fig. 1. Currently it uses a Digilent Spartan-3 board with a Spartan-3-400 Xilinx FPGA. For sensing the environment, this work employs a sonar sensor along with shaft encoders to measure movement.

Fig. 1 - Robot

This paper is organized as follows. Section 2 presents the mapping problem. Section 3 presents the robot and the main system architecture. Section 4 shows the algorithms. Sections 5 and 6 present experimental results for two embedded architectures, and finally section 7 concludes this paper.

2. Mapping in Robotics

The problem of mapping can be defined as the process of building a spatial model of the environment of a robot through the perception of the world by means of sensors. It is often referred to by the navigation question *Where have I been?*[7].

Current algorithms applied to robotics are mainly probabilistic due to uncertainty and noise of the sensors they depend on.

To perform the mapping task it is often necessary to define a representation of the environment. A common technique developed by Elfes and Moravec in the 80's is characterized by the use of occupancy grids together with known posture information about the robot (location and orientation) [7,8]. An occupancy grid is a matrix of cells where each one may be busy or free, and is a data structure used to integrate data from multiple sensors in a model of the environment. A representation of an occupancy grid is shown in Fig. 2. The integration of sensor data is performed using probabilistic algorithms based on a theory of evidence [7].

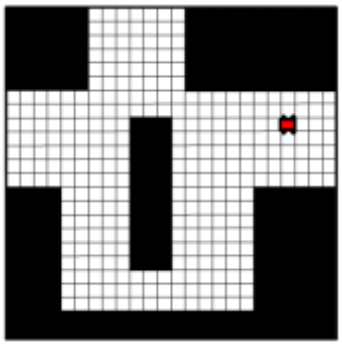


Fig. 2 - Example of an occupancy grid

The maps based on occupancy grids are not completely accurate due to incomplete data or noisy sensors. There is much ambiguity in the data collected by the sensors. The most common applications of this technique use sensors, such as sonars or laser range finders, to detect obstacles and distances.

The sonar model covers an entire cone in space (see Fig. 3). Obstacles may be found somewhere in the range of the cone. In addition to being characterized by noise, sonars suffer from the effects of reflective material obstacles, which depends upon the angles at which they stand in relation to the beam and to the other obstacles.

Mapping is based on successive readings. Each reading from a sensor is compared with the map, changing the probability of target cells. This procedure assigns to each cell the probability of being occupied.

The final maps are probabilistic and are drawn from Bayes's filters, in its simplest version. This technique is robust and easy to implement. However, it fails to address explicitly the pose of the robot.

Another shortcoming of occupancy grids is inherited from Bayes's filter, which assumes independent noise. Finally, the assumption of independence between cells constitutes another weakness of this approach, since it seldom occurs in practice [7].

The occupancy grid is combined with a model of sensor uncertainty for updating the probability values. The values are again updated as the robot navigates through the environment.

Sonar sensors are commonly used in the construction of environment maps as occupancy grids. The simplest sensor model for a sonar projects a cone in space characterized by an angle of view **β** that represents half the cone angle and by the distance **R** that defines the maximum extent of the beam (see Fig. 3). The cone is then overlapped in the occupancy grid and is divided into four regions:

a) **Region I**: Where the affected cells are probably occupied;
b) **Region II**: Where the affected cells are probably empty;
c) **Region III** and **IV**: Where the condition of the cells is unknown.

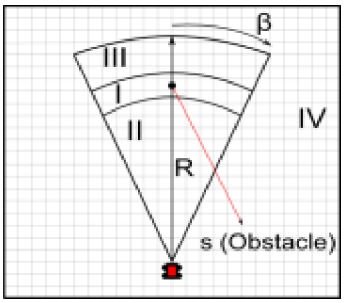


Fig. 3 - Uncertainty sensor model (sonar)

The probability values of the cells are better interpreted using probabilistic methods of data fusion from multiple sensors into a single occupancy grid, such as the Bayesian and the Dempster-Shafer methods [7]:

a) **Bayes:** In this method, the sensor model generates conditional probabilities of the form $P(s|H)$, which are subsequently converted into $P(H|s)$ by Bayes' rule, where H stands for hypotheses and s for a new reading from the sensor. The application of this theory allows two readings, whether in the same instant of time or in different instants of times, to be easily combined. Each position in an occupancy grid is associated with the probabilities $P(Empty|s)$ and $P(Occupied|s)$.

b) **Dempster-Shafer:** instead of measuring the probability of the proposition, this method

978-972-789-304-1 REC'2010

measures the *belief mass* by means of a *possibilistic* belief function, which is the way this theory represents evidence. Each new reading from a sensor, contributes with a *belief mass* (can be understood as a weight) which is distributed in a variety of combinations to the propositions. A belief function to represent an occupancy grid has a tuple with three members and can be written as:

$$Bel = m(Occupied), m(Empty), m(DontKnow)$$

When the sensor returns an ambiguous reading, it will allocate fraction of its *mass* to the term *dontknow*.

The most popular method for combining functions of possibility is the Dempster's Rule of Combination, also called Orthogonal Sum [7].

3. Robot and Main System Architecture

The infrastructure of the robot is based on the VEX Robotics kit [6].

The main architecture comprises a connection between the FPGA platform and the VEX control unit. The first choice of architecture for our robot is shown in

Fig. 4 and named herein as Architecture #1. The FPGA has embedded the primary control unit of the robot, i.e., the embedded system running the control and mapping routines. It issues commands to the slave VEX control unit, while receiving data from the sensors connected to the slave control unit.

The complete architecture is thus composed by two control units:

a) **Primary control unit:** Embedded in the FPGA platform, the Microblaze microprocessor runs a program that invokes functions that communicate with the general purpose input/output (GPIO) to send commands to the slave control unit and receive sensor data from it.

b) **Slave control unit:** The VEX control unit acts as a slave to the FPGA embedded system, providing sensor data to it and actuating on the motors according to the commands it receives from the primary unit.

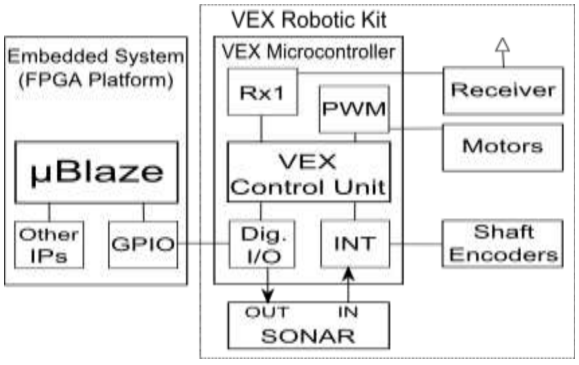


Fig. 4 - Architecture #1

According to the datasheet of the VEX controller, it updates data for the motors each 18.5*ms*. In that period, it also interprets the signals sent by the remote control, if enabled. This time frame serves as a reference for the time that is available to perform all the operations necessary to map an environment:

a) transfer data between control units (motor commands and sensors data) ;
b) update occupancy grid using sensor models and the sensor data from the previous reading;
c) compute commands for the motors;
d) restart sensors for new reading .

In Architecture #1 we encountered several bottlenecks. The first and most obvious one is related to the implemented algorithms, specifically the update of the occupancy grid given new data from the sonars and shaft encoders. It should complete the calculations in the least amount of time so that the other operations enumerated before can be completed in 18.5*ms*. Another bottleneck is related to the data transfer rate between the two control units. The primary unit operates at a higher frequency than that of the slave unit, making it wait for a reply when transferring data, thus keeping it locked at that function and preventing it from executing other tasks.

The activation of the sonar, subsequent handling of its data and the operation of the motors are done with a delay. That is, after the mapping has been successfully done and the commands have been issued, they will only be updated every 18.5*ms* at most. Thus, it is important that all the operations are performed within this amount of time.

To overcome these bottlenecks, an alternative architecture was proposed, herein addressed as Architecture #2 and displayed in Fig. 5. The sensors are now connected to the FPGA Platform allowing a more precise control and requiring less time to obtain their data. To control the sensors, customised IP cores were developed. This solution also makes the embedded system more flexible, allowing more sensors to be connected, limited only by the physical constraints of the FPGA platform (e.g., free area and/or number of pins).

Now, the embedded system only issues commands to the VEX control unit in a one-way communication. The data retrieved from sensors are handled by the associated IP cores and accessed by the main program when needed.

The new approach leads us to the conclusion that the main bottleneck that remains is related to updating the occupancy grid algorithm. This architecture is quite immune to all other bottlenecks that affected Architecture #1.

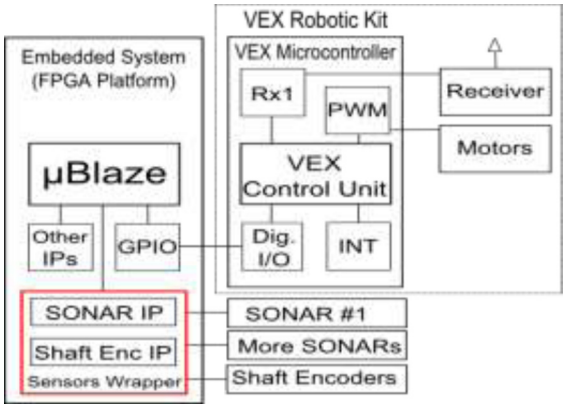


Fig. 5 - Architecture #2

4. Mapping Algorithms

The core routines needed to implement the formal theories of evidence previously discussed are herein presented. Before beginning to fill the cells of the occupancy grid with updated probabilities, it is necessary that each cell is initialized according to the chosen theory of evidence: Bayes or Dempster-Shaffer. This step is described in Routine 1.

```
1.  for each cell do:
2.      reset probabilities according to theory of evidence
```

Routine 1 - Resetting cell probabilities

The first approach to update cell probabilities during the mapping task is illustrated in Algorithm 1.

```
1.  Find vector with α=0
2.  normalize distance to object
3.  for each cell do:
4.      find new vector related to cell
5.      compute inner product
6.      compute inner angle
7.      if (inner angle < β) then
8.          compute distance to cell
9.          if ((s - Tolerance) < distance < (s + Tolerance)) then
10.             compute cell probabilities considering Region I
11.             update cell probability
12.         else if (distance < (s + Tolerance)) then
13.             compute cell probabilities considering Region II
14.             update cell probability
15.         else
16.             leave cell as is
```

Algorithm 1 – Updating cell probabilities

Given a valid reading of a distance to an obstacle *s*, the standard operating method for Algorithm 1 computes a direction vector with α=0º, connecting the cell which holds the current absolute position of the robot and the (normalized) cell in which the obstacle is supposedly in, where α is an angle between –β and β. This vector is used to calculate the inner angle with each vector connecting every other cell in the occupancy grid. If the inner angle, relating to α, is inferior to |**β**| and within boundaries of Region I and/or II, then the probabilities for that given cell are updated.

An obvious optimization to this implementation considers a sub-grid adjusted to contain the cone related to a new reading *s*. This approach reduces significantly the number of unwanted calculations related to cells that are not inside the sensor model cone. Line 3 of Algorithm 1 becomes as follows:

```
3.  for each cell within the sub-grid do:
```

The results from the previous implementations lead us to believe that further optimizations to the algorithm could be made. In light of this belief, the second approach creates a pre-filled cone and rotates it whenever a new *s* is given to match the new direction vector. A representation of this principle is visible in Fig. 6.

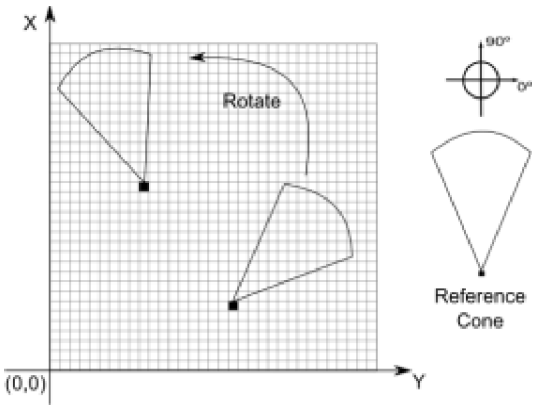


Fig. 6 - Rotation principle

Routine 2 creates a static model cone holding for each position (*x, y)* the probabilities for the chosen theory of evidence.

```
1.  Find vector with α=0
2.  normalize distance to object
3.  for each cell within the sub-grid do:
4.      find new vector related to cell
5.      compute inner product
6.      compute inner angle
7.      compute distance to cell
8.      if ((inner angle < β) and (distance < RANGE))then
9.          staticCone.position = cell
10.         staticCone.p(occupied) = probability from Region I
11.         staticCone.p(empty) = probability from Region II
```

Routine 2 - Create a cone with probabilities

Routine 2 is the complementary step necessary to use Algorithm 2 as the main algorithm to populate cells with probabilities.

This second approach rotates counter clockwise the reference cone built by Routine 2 by using a

transformation matrix to compute the cells where probabilities need to be updated. This transformation matrix is represented as follows:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad (1)$$

which gives:

$$\begin{aligned} x' &= x\cos\theta + y\sin\theta \\ y' &= -x\sin\theta + y\cos\theta \end{aligned} \quad (2)$$

Algorithm 2 presents the pseudo-code to implement this form of updating probabilities.

```
1.  while (reference cone.y  < (s + Tolerance)):
2.      compute absolute x and y using the reference cone
3.      if (cell is inside boundaries) then
4.          if (reference cone.y >= (s - Tolerance)) then
5.              compute cell probabilities considering Region I
6.          else if (distance < (s + Tolerance)) then
7.              compute cell probabilities considering Region II
8.          Else
9.              leave cell as is
10.
11.         update cell probability
```

Algorithm 2 – Update cells using reference cone

This algorithm is visibly more efficient than the previous one. It uses only the positions in the reference cone, thus it does not make unneeded calculations for cells that are not used.

Comparing to Algorithm 1, the apparent bottleneck in Algorithm 2 is the step needed to rotate coordinates in line 2. This is because of the inherent mathematical operations. The rest of the algorithm remains almost unaltered.

In order to lower the weight of this operation we have replaced the calls to sine and cosine functions with look-ups of values stored in tables. These are either predetermined arrays with *sine* and *cosine* values or arrays filled in the beginning of the main program before any other procedure.

If we choose to fill an array then after Routine 1 another routine (see Routine 3) needs to be called to create an array that contains *sine* and *cosine* values (in this implementation we use array values for angles between 0º to 360º).

```
1.  for angle=0 to angle=360 do:
2.      sinearray[angle] =sine(angle)
3.      cosinearray[angle] =cosine(angle)
```

Routine 3 - Create tables for sine and cosine values

5. Experimental Results (Arch. #1)

A study was conducted to evaluate the performance of mapping algorithms when using an embedded system containing a MicroBlaze operating at 50MHz implemented on a Digilent Spartan 3 XC3S200 FPGA development board. We followed a methodology that evolved from the worst possible case to the best.

We started with an implementation that runs through all the cells of the matrix regardless of its size.

All the initial tests were performed with the implementation of Bayes' theory of evidence. We started with a matrix of dimensions 100×100 cells, initialized it and updated it once considering an obstacle at an average distance of 6 units.

It was expected that on the embedded system the mere conversion of double to float precision would substantially improve results. This indeed happened. Profiling results (see Fig. 7) showed a reduction in execution time to about half, by compiling with optimization O2 (696.08s with doubles in contrast to 335.00s with floats). Compiling with optimization O3 both times were improved. As expected, single precision (float) still got better results. Execution time was shortened approximately by 1.8 times (335 s using O2 contrasting to 186.9 s using O3).

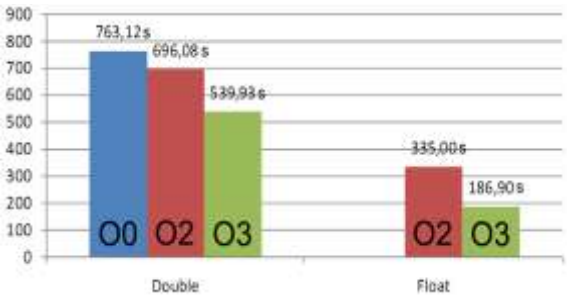


Fig. 7 - Profiling times for considering data type optimization (Option #1)

Having obtained better results with O3 compilation, we proceeded to evaluate execution performance with extra hardware modules included in the design (e.g., FPU and Cache memory). In addition to the results obtained by profiling, we used hardware timers to measure the real execution time of each function needed for the different settings.

Considering the different evolutions of the updating algorithms, we hereby summarize them in a list for further reference:

Option #1 – Algorithm 1 testing each cell in the occupancy grid;

Option #2 – Algorithm 1 testing each cell in a sub grid containing the cone;

Option #3 – Algorithm 2 rotating a reference cone using a transformation matrix.

With the inclusion of additional modules and single precision data types, we present real execution times for **Option #1** and **#2** in Table 1.

Of the three hardware solutions that were presented, the one which used Cache memory

performed better in the overall tests obtaining results about 7 times faster than their counterpart without hardware extensions.

Hardware	Option #1 (s)	Option #2 (s)
Base	74.88	0.22
with FPU	11.55	0.07
with Cache	11.03	0.03

Table 1 - Execution times (in seconds) comparison with hardware modules for Bayes's implementation

The overall occupancy for the FPGA used for testing was 93% for the simplest solution and about 99% for both solutions with additional hardware components. This fact did not allow us to perform tests using both FPU and Cache memory, so we tested the prototype with the hardware module that, up to this point, showed to get better results. So, from this point onwards, the results reflect an architecture where only cache memory is used, for both mapping algorithms being analysed – Bayes and Dempster-Shaffer.

Table 2 shows the evolution of execution time of the algorithms as a function of the size of the occupancy grid for Option #1 and Option #2, and for both implementations of theories of evidence.

	Bayes		Dempster-Shaffer	
Grid size	**Option #1 (s)**	**Option #2 (s)**	**Option #1 (s)**	**Option #2 (s)**
32×32	1.1074	0.0705	2.0597	0.1208
50×50	2.7299	0.0708	5.1108	0.1213
64×64	4.4947	0.0712	8.4337	0.1219
100×100	11.0313	0.0726	20.7605	0.1240
128×128	18.1133	0.0759	34.2249	0.1269
150×150	24.9080	0.0756	46.9562	0.1285

Table 2 – Evolution of execution time (in seconds) as grid size is increased

We initialized and updated the occupancy grid once using an obstacle at an average distance of 6 units for different grid sizes. From the resulting values, it is apparent that execution times increase almost exponentially as the size of the occupancy grid increases. Dempster-Shaffer's implementation is computationally more demanding and thus grows faster than Bayes's implementation.

With respect to Option #2, we can observe that the test routines have a similar performance for different sizes of the grid. In fact, for an increase of about 40 times the number of cells there is only a slight increase of 1*ms* in the execution time. Dempster-Shaffer's implementation remains the most computationally demanding, taking almost twice as much time as Bayes' implementation. The few milliseconds increase lead us to believe that it is due to the initialization of the grid, since the updating of the occupancy grid is related, not to the size of the grid but to the distance to an object.

Without counting with the initial steps that initialize the occupancy grid, we have obtained minor differences in results for the grid sizes considered. We obtained values close to 0.0702 s and 0.1204 s for Bayes's and for Dempster-Shaffer's implementations, respectively. Thus, Option #2 is not dependent on the size of the occupancy grid, but on the distance to an object. Keeping that in mind, further tests were conducted to determine the evolution of execution time related to different distances to an object (see Table 3).

Distance (no. of cells)	Bayes (ms)	Dempster-Shaffer (ms)
5	12.23	17.01
6	27.41	44.26
7	32.39	52.72
8	37.40	61.23
9	42.93	70.30
10	70.20	120.40

Table 3 – Evolution of execution time (in milliseconds) as distance is increased for Option #2

From these results, it can be observed that Dempster-Shaffer's implementation remains computationally more demanding than Bayes' implementation, and that Option #2 is actually only dependent on the distance to an obstacle.

We performed some experiments in the field with our prototype. The robot should deviate from the obstacles and perform the mapping of the environment. The occupancy grid consists of 30×30 units, the equivalent to 9m^2. The robot considers the starting position in the centre of the map (x = 15 and y = 15, in this example). In Fig. 8, we show the environment with the dashed line representing the path actually traken by the robot. This test used a random selection of turns whenever the robot faced an obstacle. The route presented required less than 10 seconds.

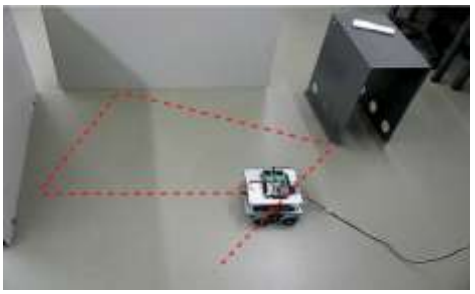


Fig. 8 - Mapping environment

The occupancy grid built by the algorithm at the end of the experiment is represented in Fig. 9. In this figure are also illustrated the areas considered as obstacles and the likely route travelled by the robot. The gray cells indicate the limits of obstacles in the environment as found by the robot.

The uncertainty is derives from the sonar model applied to the map. In Region I cells are considered to be probably occupied and in Region II as probably empty. The fusion of multiple reads of the same cell does vary the odds associated with that cell, hence the range of probability values in the vicinity of an obstacle.

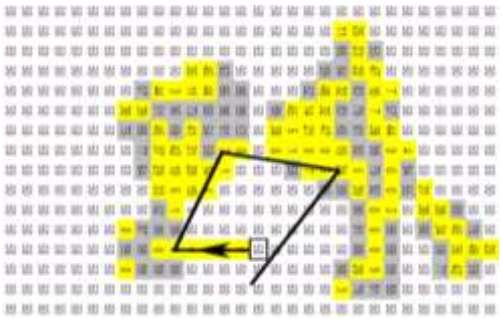


Fig. 9 - Occupancy grid as updated by the robot

6. Experimental Results (Arch. #2)

In order to improve the results obtained with Architecture #1, a few changes were made. Architecture #2 is the result of those changes. It was implemented on a Spartan 3 XC3S400 FPGA Platform with a 50MHz clock. This architecture has some advantages over the previous one. The current FPGA now has capacity for both FPU and Cache memory. Remember that XC3S200 chip could not have both hardware modules in the same design. The results considering these two hardware modules are displayed in Table 4 for both theories of evidence.

Distance (no. of cells)	Bayes (ms)	Dempster-Shaffer (ms)
5	7.20	17.58
6	14.80	52.24
7	17.75	62.89
8	20.69	73.53
9	24.25	84.84
10	36.95	148.84

Table 4 - Evolution of execution time (in milliseconds) as distance is increased (with FPU and Cache memory for Option#2)

Comparing with Table 3 we notice an increase in performance of almost 2 times for Bayes' implementation. For Dempster-Shaffer's implementation the execution time increased as much by 20% (120.4 ms shown in Table 3 comparing to 148.84 ms on the current design).

Next we considered some improvements of the updating algorithm. Using Option #3, we conducted the same tests as before obtaining the results displayed in Table 5. The new algorithm's execution time is still dependent on the distance to an object.

Distance (no. of cells)	Bayes (ms)		Dempster-Shaffer (ms)	
	O2	O3	O2	O3
5	4.48	4.80	5.71	4.81
6	5.96	6.40	7.63	6.43
7	7.45	8.00	9.54	8.04
8	10.01	10.73	12.78	10.79
9	12.47	13.40	15.96	13.44
10	12.48	13.40	16.00	13.45

Table 5 – Comparison of execution times (in milliseconds) for Option #3

The new algorithm, along with *float* data types, FPU and Cache memory, provides better results. There is an increase in performance of nearly 2 times for Bayes' implementation, obtained using the new architecture with Option #2 algorithms. For Dempster-Shaffer, the increase in performance is clear, ranging from 3 times for a distance of 5 units to 11 times better for a distance of 10 units using compiler optimization O3. In architecture #1 the difference between execution times of both implementations almost excluded the Dempster-Shaffer implementation as it took too long, but using this new algorithm it becomes competitive again.

Although good improvements have been made, we tried to make further improvements so that the system could cope with several sensors updating the grid in a given interval of time.

After careful study of Algorithm #2 it is noticeable that the main bottleneck could be caused by the rotation operation in line 2, which involves several mathematical operations with real numbers. This is the mathematical operation within the algorithm that is easy to alter, as the other operations are related to the theory of evidence in use. Note that *sine* and *cosine* operations are now based on lookup tables. In the *sine* and *cosines's* tables instead of storing their real values, this new version stores their fixed-point representation. We add the following option to our range of tested algorithms and optimizations:

Option #4 – Algorithm 2 rotates a reference cone using a transformation matrix and each cell of the cone contains the corresponding values for occupied/empty probabilities stored in a structure. It uses values in a fixed-point representation for some computations.

In Table 6 we compare both execution times for **Option #3** and **#4**. The last optimization to the algorithm was successful in obtaining even better results. The execution time of Bayes's implementation took up to 3*ms* now.

Distance (no. of cells)	Optimization O2 (ms)		Optimization O3 (ms)	
	#3	#4	#3	#4
5	4.48	3.97	4.80	3.59
6	5.96	5.28	6.40	4.76
7	7.45	6.58	8.00	5.93
8	10.01	8.79	10.73	7.90
9	12.47	10.95	13.40	9.83
10	12.48	10.96	13.40	9.83

Table 6 – Comparison of execution times (in milliseconds) for Option #3 and #4 with Bayes's implementation

Table 7 shows the results for Dempster-Shaffer's implementation reducing also up to 3*ms* in execution time, and remaining competitive with Bayes's solution.

Distance (no. of cells)	Optimization O2 (ms)		Optimization O3 (ms)	
	#3	#4	#3	#4
5	5.71	4.84	4.81	3.95
6	7.63	6.46	6.43	5.25
7	9.54	8.07	8.04	6.54
8	12.78	10.79	10.79	8.73
9	15.96	13.45	13.44	10.86
10	16.00	13.49	13.45	10.86

Table 7 – Comparison of execution times (in milliseconds) comparison between Option #3 and #4 for Dempster-Shaffer's implementation

7. Conclusions

This paper proposed a mobile robot coupled to a reconfigurable computing platform. The reconfigurable computing platform consists of an FPGA board and is responsible to acquire sensor data, to control de robot, and to execute navigation algorithms. Our first implementations consider two mapping algorithms able to build a map of the environment based on data acquired from a sonar.

This paper showed how the addition of extra hardware modules and optimizations over the two mapping algorithms can improve substantially the execution time of a real-time embedded system implemented on FPGAs. All the improvements performed in this system lead to an overall gain in real time processing. Implementation from Bayes's algorithm, at an average distance of 6 units, improved from 74.88s using a very naive implementation which had no optimizations to the algorithms and used double data types (Option #1), to the 4.76ms using with float data types, inclusion of hardware modules and relevant optimizations of the algorithms (Option #4). Dempster-Shaffer's implementation followed the same pattern, ending up updating the occupancy grid in 5.25ms at 6 units distant using Option #4.

The optimizations performed are important to obtain a shorter control loop and/or to allow the use of more sonars in order to improve the mapping capabilities without slowing down the robot.

Acknowledgments

This work has been partially funded by FCT (*Fundação para a Ciência e a Tecnologia*) under grant PTDC/EEA-ELC/70272/2006.

References

[1] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surv.*, vol. 34, 2002, pp. 171-210.

[2] G. Rubin, M. Omieljanowicz, and A. Petrovsky, "Reconfigurable FPGA-Based Hardware Accelerator for Embedded DSP," *Mixed Design of Integrated Circuits and Systems, 2007. MIXDES '07. 14th International Conference on*, 2007, pp. 147-151.

[3] Z. Salcic, "PROTOS-- A microcontroller/FPGA-based prototyping system for embedded applications," *Microprocessors and Microsystems*, vol. 21, Dec. 1997, pp. 249-256.

[4] D. Wolf, J. Holanda, V. Bonato, R. Peron, and E. Marques, "An FPGA-Based Mobile Robot Controller," *Programmable Logic, 2007. SPL '07. 2007 3rd Southern Conference on*, 2007, pp. 119-124.

[5] Min Xu, Wenzhang Zhu, and Ying Zou, "Design of a Reconfigurable Robot Controller Based on FPGA," *Embedded Computing, 2008. SEC '08. Fifth IEEE International Symposium on*, 2008, pp. 216-222.

[6] "VEX Robotics Design System," *http://www.vexrobotics.com/*.

[7] R.R. Murphy, *An Introduction to AI Robotics*, The MIT Press, 2000.

[8] S. Thrun, "Robotic mapping: a survey," *Exploring artificial intelligence in the new millennium*, Morgan Kaufmann Publishers Inc., 2003, pp. 1-35.

Sessão Regular 6

Telecomunicações II

Moderação: Luís Gomes
Universidade Nova de Lisboa / UNINOVA

Implementação em FPGA de um desmodulador DCM para um receptor UWB MB-OFDM

Hugo Santos[†], Mário Véstias[†], Helena Sarmento[‡]
[†]*INESC-ID/ISEL/IPL,* [‡]*INESC-ID/IST/UTL*
28403@alunos.isel.ipl.pt, mvestias@deetc.isel.ipl.pt, hsarmento@inesc-id.pt

Resumo

Este artigo apresenta a implementação de um desmodulador DCM e a sua integração num receptor UWB MB-OFDM. O receptor foi descrito usando MATLAB/Simulink. A partir desta descrição, utilizou-se o Xilinx System Generator para gerar o VHDL do desmodulador. O circuito foi simulado em Simulink integrado com o receptor UWB e posteriormente foi cosimulado usando uma placa ML402 com uma FPGA Virtex-4 da Xilinx. O circuito cumpre todos os requisitos temporais e ocupa apenas cerca de 3% dos recursos da FPGA[1].

1. Introdução

O UWB (*Ultra-Wideband*) MB-OFDM (*Multiband Orthogonal Frequency Division Multiplexing*) é uma tecnologia sem fios de curto alcance. A tecnologia está especificada no ECMA-368, 3ªedição [1]. O UWB MB-OFDM usa o espectro de frequências situado entre 3.1 GHz e 10.6 GHz. O espectro está dividido em 14 bandas, cada uma com uma largura de banda de 528 MHz que pode suportar até 480 Mbps. Dependendo da modulação e da taxa de codificação usada, o sistema suporta ritmos de dados de 53.3 Mb/s, 80 Mb/s, 106.7 Mb/s, 160 Mb/s, 200 Mb/s, 320 Mb/s, 400 Mb/s e 480 Mb/s, como é observado na tabela 1, em que R representa a taxa de codificação do sistema.

Ritmo de Dados	Modulação	R
53.3 Mbps	QPSK	1/3
80 Mbps	QPSK	1/2
106.7 Mbps	QPSK	1/3
160 Mbps	QPSK	1/2
200 Mbps	QPSK	5/8
320 Mbps	DCM	1/2
400 Mbps	DCM	5/8
480 Mbps	DCM	3/4

Tabela 1. Parâmetros de funcionamento do MB-OFDM

Um símbolo OFDM é composto por 128 portadoras (números complexos). Estas dividem-se em 100 sub-portadoras de informação, 10 de guarda, 12 piloto e 6 de enchimento. As 10 de guarda são usadas para evitar as interferências inter-símbolos. As 12 piloto permitem uma detecção mais coerente, introduzindo robustez contra as frequências que estão fora do alinhamento (*offset*) e ruído de fase. As sub-portadoras de enchimento são aplicadas para evitar interferências por multi-caminho.

O espaçamento das sub-portadoras é de 4.125 MHz, sendo igual à frequência de amostragem (528 MHz) a dividir pelo número total de sub-portadoras (128). O UWB MB-OFDM é assim uma tecnologia aplicada a rede sem fios, fazendo parte do grupo de redes WPAN (*Wireless Personal Area Network*), sendo uma tecnologia bastante eficiente em curto alcance possibilitando altos débitos.

Este artigo apresenta uma implementação hardware do desmodulador DCM (*Dual Carrier Modulation*) para um receptor UWB MB-OFDM. O sistema foi inicialmente modelado no Simulink da ferramenta de trabalho MATLAB 7.6.0 (R2008a), a partir do qual se verificou o seu comportamento com base em simulações e co-simulações. As simulações são efectuadas através do Simulink, onde à excepção do bloco desmodulador, todo o modelo é composto por blocos da biblioteca Simulink. As cosimulações são efectuadas através do Simulink e do gerador de sistemas da ferramenta Xilinx System Generator 10.1. Com o uso do gerador de sistemas gerou-se uma descrição VHDL do desmodulador DCM a partir dos blocos da Xilinx e de seguida procedeu-se a uma cosimulação de todo o bloco numa FPGA (Virtex-4 vsx35-10ff668).

O artigo está organizado como se descreve de seguida. A secção 2 descreve a desmodulação DCM. No capítulo 3 apresentamos a arquitectura proposta para o desmodulador DCM. Na secção 4 é descrito o modelo Simulink usado no teste do desmodulador DCM. Na secção 5 são apresentados os resultados de simulação, cosimulação e de síntese em FPGA. Por fim, na secção 6 são apresentadas as conclusões e o trabalho futuro.

2. Modulação DCM

O modulador DCM é apenas utilizado nos 3 ritmos mais elevados de comunicação de dados: 320 Mbps, 400 Mbps e 480 Mbps (Ver tabela 1).

O modulador DCM recebe 100 valores complexos, representando 200 bits. Estes valores são divididos em 50 grupos de 4 bits. Cada grupo de 4 bits é representado pela equação (1).

[1]This work has been performed under the project "UWB Receiver: baseband processing using reconfigurable hardware- PTDC/EEA-ELC/67993/2006

$$(b[g(k)], b[g(k)+1], b[g(k)+50], b[g(k)+51]) \quad (1)$$

em que g(k) é dado por:

$$g(k) = \begin{cases} 2k & \text{if k} \in [0,24], \\ 2k+50 & \text{if k} \in [25,49]. \end{cases}$$

Cada grupo de 4 bits é convertido em dois números complexos d[k] e d[k+50], em que d[k] = I[k] + Q[k]i e d[k+50] = I[k+50] + Q[k+50]i. Os valores I[k], I[k+50], Q[k] e Q[k+50], representados na tabela 2, variam consoante os 4 bits de entrada do modulador.

Bits de entrada	I[k]	Q[k]	I[k + 50]	Q[k + 50]
0000	-3	-3	1	1
0001	-3	-1	1	-3
0010	-3	1	1	3
0011	-3	3	1	-1
0100	-1	-3	-3	1
0101	-1	-1	-3	-3
0110	-1	1	-3	3
0111	-1	3	-3	-1
1000	1	-3	3	1
1001	1	-1	3	-3
1010	1	1	3	3
1011	1	3	3	-1
1100	3	-3	-1	1
1101	3	-1	-1	-3
1110	3	1	-1	3
1111	3	3	-1	-1

Tabela 2. Tabela de Mapeamento DCM

Para o desmodulador DCM usou-se o algoritmo proposto por Yang e Sherratt [2]. Este algoritmo usa os valores complexos das sub-portadoras k e k+50 de forma a obter uma representação dos bits de entrada do modulador DCM. O algoritmo usa ainda um CSI (*Channel State Information*). O CSI é uma estimação dinâmica do canal e é usado para melhorar o resultado do desmodulador na presença de ruído térmico. Excluindo o ruído térmico (e consequentemente o CSI) e as interferências por multicaminho, a representação dos bits de entrada é dada pelas equações (2-5) [3].

$$b_{g(k)} = 2I_{R(k)} + I_{R(k+50)} \quad (2)$$

$$b_{g(k+1)} = I_{R(k)} - 2I_{R(k+50)} \quad (3)$$

$$b_{g(k)+50} = 2Q_{R(k)} + Q_{R(k+50)} \quad (4)$$

$$b_{g(k)+51} = Q_{R(k)} - 2Q_{R(k+50)} \quad (5)$$

3. Desmodulador DCM Proposto

De forma a conseguir processar os dados a uma velocidade suportada pela FPGA alvo, é proposto um bloco desmodulador com dois módulos de desmodulação a funcionar em paralelo.

A FFT gera 100 números complexos em 242 ns, que são enviados sequencialmente para o desmodulador DCM. Assim, o bloco DCM tem de processar cada um dos 100 números complexos em aproximadamente 2.42 ns, o correspondente a uma frequência de 413 MHz.

Com o bloco DCM proposto, consegue-se colocar à saída do bloco os primeiros 200 valores em apenas 75 ciclos relógio. Na figura 1 é mostrado o diagrama de blocos do desmodulador proposto.

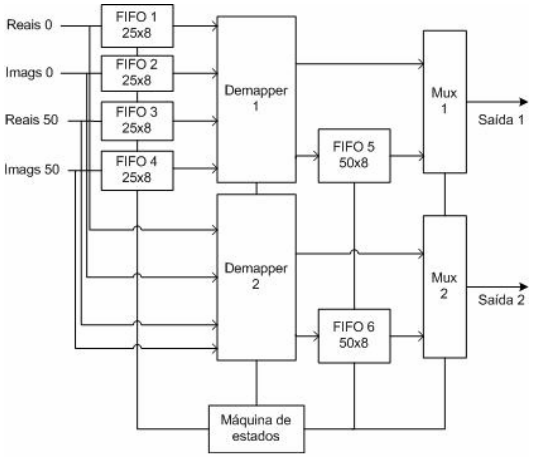


Figura 1. Diagrama de blocos do desmodulador DCM proposto

O circuito é constituído por 4 FIFO à entrada, dois módulos para o cálculo dos bits desmodulados (ver equações 2-5), duas FIFO à saída e dois multiplexers para serializar os bits desmodulados. O objectivo de cada bloco *demapper* é colocar à saída a representação dos 100 bits de entrada de acordo com as equações (2-5).

O circuito recebe 100 números complexos em série vindos da FFT em dois grupos. Um grupo contém os números complexos de 0 a 49 (I[k] + jQ[k]) e o outro de 50 a 99 (I[k+50] + jQ[k+50]). O primeiros 50 números complexos (0 to 24 and 50 to 74) são guardados em quatro FIFO de entrada para serem processados por um *demapper*. Os outros são enviados para segundo *demapper*. Quando o segundo *demapper* começa a receber os dados, ambos os *demappers* começam o processamento dos seus dados. Cada um dos *demappers* gera 100 *soft bits* a serem enviados para o descodificador de Viterbi. Uma vez que os *soft bits* não são gerados por ordem, enquanto que os primeiros 50 *soft bits* produzidos são enviados de imediato ao quantificador, os segundos 50 são armazenados nas FIFO de saída para serem enviados após os primeiros.

Cada um dos *demappers* retorna os quatro bits $b_{g(k)}$, $b_{g(k)+1}$, $b_{g(k)+50}$ e $b_{g(k)+51}$, por cada par de complexos recebido à entrada. Os quatro bits terão depois de ser agrupados em dois pares: $b_{g(k)}$ com $b_{g(k)+1}$ e $b_{g(k)+50}$ com $b_{g(k)+51}$. Contendo duas saídas representando os bits b0 a b49 e outra de b50 a b99. Assim foi decidido que à medida que os primeiros 50 valores fossem sendo processados iriam sendo libertados, enquanto que os 50 últimos eram guardados numa memória FIFO. Para não haver sobreposição na última memória (onde são guardados os últimos 50 valores), é necessário que haja um tempo em que o DCM não receba valores de forma a conseguir libertar os 100 da-

978-972-789-304-1 REC'2010

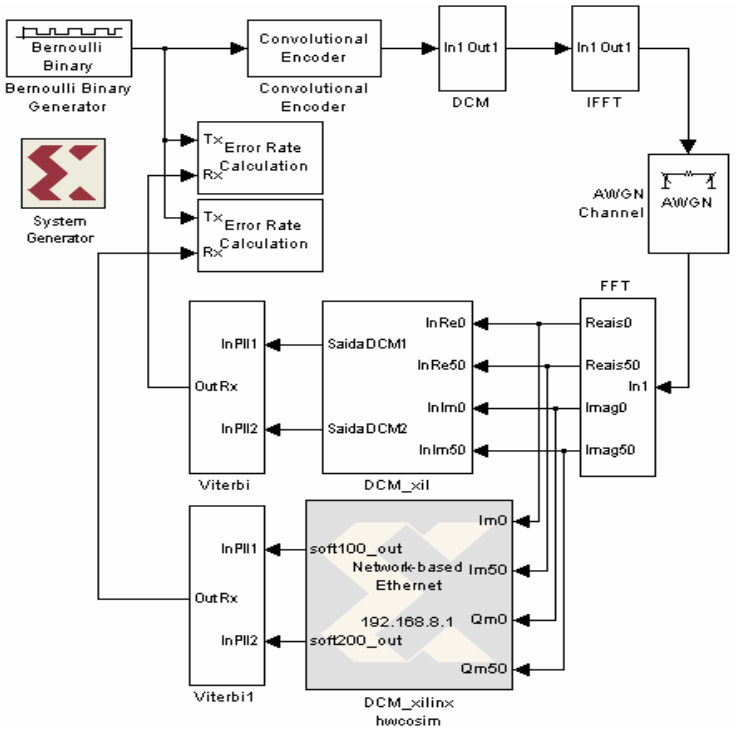


Figura 2. Modelo do sistema de teste do desmodulador

dos. Daí a utilização de FIFO. As duas saídas do desmodulador são depois tratadas em paralelo pelo descodificador de Viterbi.

4. Modelo Testado

Na figura 2 é apresentado o modelo com que o bloco DCM foi testado. Todos os blocos à excepção do bloco desmodulador DCM são compostos por sub-blocos da biblioteca Simulink.

O sistema começa por criar bits aleatórios de dados, que depois passam pelo codificador convolucional que define a taxa de codificação do sistema, podendo ser 3/4, 5/8 e 1/2. O bloco DCM efectua a modulação digital de acordo com o que já foi referido anteriormente, retornando sinais complexos. O bloco IFFT para além de acrescentar as restantes sub-portadoras, de guarda, piloto e de enchimento, efectua a IFFT às 128 sub-portadoras. No canal de transmissão é colocado um bloco AWGN para efeito de teste, variando a relação sinal ruído, SNR (*Signal Noise Ratio*).

No lado do receptor a FFT efectuará o processo oposto ao que foi feito no bloco IFFT. Passa o sinal recebido para o domínio da frequência e retira apenas as sub-portadoras de dados para que estas possam ser analisadas no bloco de desmodulação e posteriormente no bloco de descodificação de Viterbi.

Na figura apresentam-se dois blocos de desmodulação. Um deles contém sub-blocos da biblioteca Xilinx e é simulado no ambiente Simulink. O outro bloco foi gerado a partir do gerador de sistemas e, para além do desmodulador DCM, contém uma interface Gigabit Ethernet que permite fazer a ligação entre a FPGA e o Matlab de modo a realizar a cosimulação com todo o sistema a correr no Simulink, excepto o desmodulador DCM que executa na FPGA. No ambiente Simulink os números são representados com 32 bits e na implementação em hardware são usados apenas 8 bits.

5. Resultados Experimentais

Para as simulações foi usado o desmodulador DCM proposto constituído unicamente por sub-blocos pertencentes à biblioteca Xilinx do Simulink. Para as cosimulações usou-se o dispositivo FPGA Virtex-4 XC4VSX35 da Xilinx integrado na placa ML402 [4].

O ficheiro de configuração da FPGA é gerado através do gerador de sistemas pertencente ao programa Xilinx System Generator 10.1. O gerador sintetiza e mapeia os vários blocos pertencentes ao desmodulador, usando a ferramenta de síntese da Xilinx (ISE), incluindo um bloco de interface entre o Matlab e a FPGA, por onde se configura a FPGA e se trocam dados entre os diversos módulos durante a cosimulação.

Os recursos usados pelo bloco desmodulador encontram-se presentes na tabela 3. Nela verifica-se que são usadas 458 *slices* e 6 BRAM para a implementação das FIFO.

LUT/Slices	BRAM	Freq
679/458	6	330 MHz

Tabela 3. Resultados pós-P&R do desmodulador DCM

O circuito opera a uma frequência máxima de 330 MHz, suficiente para garantir o processamento dos dados provenientes da FFT em menos de 2.42 ns, uma vez que temos dois desmoduladores a funcionar em paralelo.

Para efectuar os testes ao bloco, efectuaram-se várias simulações e cosimulações do modelo considerando como entrada bits gerados aleatoriamente, um canal AWGN, codificação com 3 *soft bits* e comprimento de descodificação de Viterbi igual a 49. Variou-se a relação sinal ruído no canal de forma a obter um gráfico com várias taxas de bits errados e consideraram-se as três taxas de codificação do DCM: 1/2, 5/8 e 3/4.

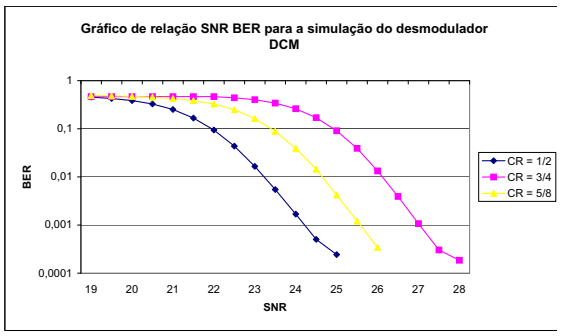


Figura 3. Simulação do desmodulador DCM

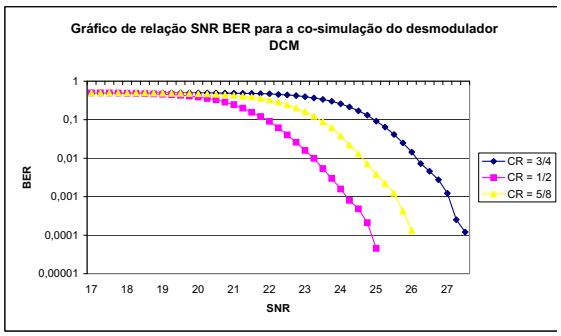


Figura 4. Co-simulação do desmodulador DCM

Na figura 3 encontra-se o gráfico referente às simulações e na figura 4 temos o gráfico referente às cosimulações.

Através das figuras, observamos que a taxa de codificação de 1/2 é a que apresenta menos erros, seguida da 5/8, como seria de esperar. Para confirmar que o desmodulador estava funcionalmente correcto, considerou-se o modelo sem ruído no canal e verificou-se que os dados recebidos eram idênticos aos enviados.

Os resultados obtidos por cosimulação são idênticos aos obtidos por simulação.

6. Conclusões e Trabalho Futuro

Este artigo descreve a implementação de um desmodulador DCM em FPGA usando o ambiente MATLAB/Simulink. O desmodulador usa cerca de 3% dos recursos de uma FPGA Virtex-4 de média dimensão e pode operar a frequências superiores a 300 MHz, suficiente para cumprir os requisitos temporais.

A FFT e o Viterbi já foram igualmente implementados em FPGA e o desmodulador DCM irá ser integrado juntamente com estes blocos para concluir a implementação do receptor UWB em FPGA.

Referências

[1] ECMA International. High rate ultra wideband PHY and MAC standard.

[2] Oswaldo Cadenas R.Yang, R. Simon Sherratt. FPGA based dual carrier modulation soft mapper and demapper for the MB-OFDM UWB platform.

[3] R. Simon Sherratt R. Yang. Dual carrier modulation demapping methods and performances for wireless USB.

[4] Inc. Xilinx. Virtex-4 family overview. September 2007.

The IEEE 802.11p Physical Layer implemented in a FPGA for the DSRC 5.9GHz project

Pedro Mar[(1)], João Matos[(1,2)], Ricardo Abreu[(1)]
[(1)]*Instituto de Telecomunicações, Aveiro, Portugal,*
[(2)]*Univ. de Aveiro, Aveiro, Portugal*
pmar@av.it.pt, matos@ua.pt, ricardo@av.it.pt

Abstract

This paper briefly describes how a Field Programmable Gate Array (FPGA) is being used in a communication system whose main goal is to improve road safety. This is accomplished by enabling each vehicle to communicate with surrounding vehicles and with fixed road side units. When an unexpected situation occurs, automatically a warning message is sent to the vehicles approaching the accident area and to the highway operator so it can call the emergency services.

The communication system, partially described here, is based on the standard IEEE 802.11p and is a vehicular communication system that uses the 5.9GHz band for Dedicated Short Range Communications (DSRC).

1. Introduction

Nowadays, car accidents constitute one of the most serious dangers each one of us faces. A significant percentage of these accidents could be avoided if, for instance, the driver could be warned of the danger several seconds before getting into the danger area.

Vehicular wireless communication systems are present in our roads today, specially in highways for toll collection. However, a vehicular communication system is a resource that should also be used to improve road safety.

Thus, the challenge is to build a communication system that can be used for toll collection and to transmit and receive warning messages as well.

For example, if an accident occurs and the airbag deploys, it would be extremely useful if a message was sent to the medium by the vehicle so the infrastructure and the following vehicles receive this warning. The infrastructure might activate the emergency services and the other drivers would know that the accident occurred before reaching the danger area.

One can think in another situation: the "hard brake". In this case, a sudden speed reduction (possibly due to hard braking) of a vehicle could be automatically detected by the system on-board, so a warning could be sent to the following vehicles within a certain range.

This kind of functions could be implemented by an on-board communication system used for toll collection as well. In this case, the infrastructure and the vehicle exchange some messages, so the identification of the vehicle is performed.

These are some examples of how important and useful road-communication systems can be.

Bearing this in mind, a prototype is being developed in a collaborative R&D project financed by Brisa and involving the Telecommunications Institute (IT), the University of Aveiro (UA) and the Engineering Institute of Lisbon (ISEL), Portugal. This project aims to be an approach to the DSRC 5.9GHz – IEEE WAVE emergent standard and service oriented solutions.

The main goal of this project is to implement a communication system based on the IEEE STD 802.11p [1] that shall contribute to the road safety as well as to the automatic tolling processing.

Regarding the importance of a system like this, we tried to make it as robust and tolerant to errors as possible.

The main focus of this document is to describe the processing performed by the circuit based on a FPGA. For simplicity we call it the Physical (PHY) layer, despite the fact that the radio-frequency (RF) blocks are not described here.

2. Adaptable bit sequence processing in PHY

As far as this document is concerned, the bit sequences constituting a single frame (thus a single message) are processed according to the rules specified by the IEEE STD 802.11p [1].

One should point that the main operations that must be performed are: scrambling, encoding and interleaving, as described in the next sub-sections and in Fig.1 and Fig.2.

However, the modulations referred in the IEEE 802.11p determine the proceedings that should be used. The encoding and interleaving operations vary according to the modulation used for the current transmission, instead of being always performed the same way.

2.1 Parameters' Managers

The operations performed and described in this document depend on the desired modulation. This could be a problem because each block does not know *a priori* which is the current modulation.

Actually, this problem is even more serious during reception, because there is no other information but the one included in the frame itself.

To solve this problem, the standard IEEE 802.11p demands that the first 48 bits (24 bits before the encoding process) of the frame shall always be transmitted using the same encoding and modulation, regardless the modulation of the other bits in the frame. To support this, a couple of blocks acting as "parameter managers" were included in the transmission chain and in the reception chain of the PHY.

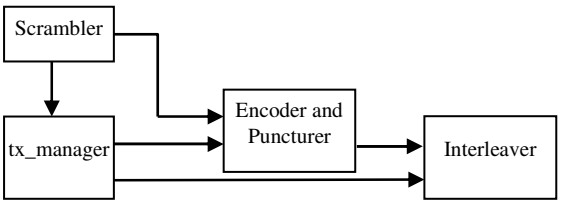


Fig. 1 – Blocks used in the transmission

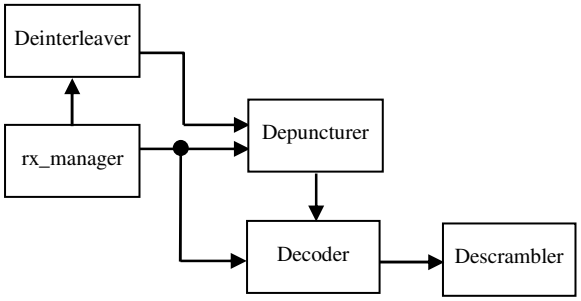


Fig. 2 – Blocks used in the reception

The management block (Fig. 1) present in the transmission chain (*tx_manager*) extracts the first four bits of the sequence because the information of the desired modulation is codified there. After this, it gives this information to the encoding, puncturing and interleaving blocks so they can process the sequence according to the rules established for that modulation. Another relevant function of this manager is to extract the length of the sequence (specified in the frame itself), count the bits outputted by the transmission chain, and generate a signal acting as an internal reset to the whole chain, in the end.

On the other hand, the reception chain (Fig. 2) handles this situation using a similar but more complex approach. To process the received sequence, its initial 48 bits are deinterleaved and decoded by a management block whose goal is to extract the first bits identifying the modulation used. Additionally, the whole bit sequence is stored to prevent the loss of bits during the time on which this is performed. After identifying the modulation used in the transmission, the manager sends to the other blocks this information as well as the stored bit sequence. This is how every block on the receiver knows which is the modulation in use so it can process the bit sequence according to those parameters.

2.2 Scrambling

The scrambling operation is used to eliminate long sequences consisting of '0' or '1', which are undesirable in a communication system as ours. This operation is known as "code whitening".

As illustrated in Fig. 3, the input of the scrambler is a bit sequence, as well as its output. Internally, it generates a pseudo-random binary sequence and adds (XOR) its value to the incoming bits, so the output bits are the result of this operation.

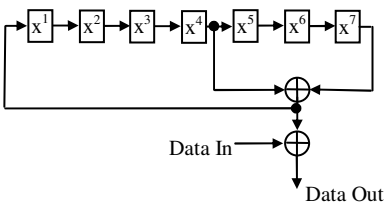


Fig. 3 – Data Scrambler

In Fig. 3 each "box" represents a bit position in a shift register and the XOR operation is shown with the usual addition sign. Thus, as is clear, this is the structure of a common pseudo-random sequence generator.

One should note that the first 48 bits (24 bits before the encoding process) of each frame are not scrambled, so the parameters "modulation rate" and "length" contained there can be easily extracted by the *tx_manager* mentioned above (Fig. 1).

2.3 Encoding and Puncturing

Every bit sequence is encoded using the convolutional encoder represented in Fig. 4, which means that a 1/2 redundancy is introduced here.

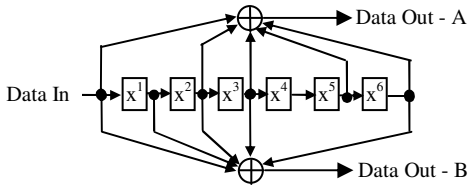


Fig. 4 – Convolutional Encoder

Regarding the modulation used to transmit a frame, the rate will be automatically set to the desired value. This means that sometimes another operation, the puncturing, shall also be performed to achieve rates of 2/3 and 3/4, besides the rate of 1/2 (no puncturing).

This is why the puncturer acts as a sub-block of the encoder in the transmission chain of the PHY.

The redundancy introduced in the transmission chain enables data recovery in the receiver even if some bits are ill-received.

When receiving the sequence, this redundancy is used and removed by the depuncturer and the decoder. Actually, this decoder is based on an ALTERA IP-Core implementing the Viterbi's Algorithm. The depuncturer was designed to generate several configuration signals to the viterbi's decoder as well as to add some null bits where necessary, regarding the modulation used in the transmission (thus reverting the puncturing performed by the transmitter).

2.4 Interleaving

The operation named "interleaving" is performed by arranging the data bits according to several specified rules named "permutations". This is used to guarantee that an error burst does not affect a group of consecutive bits. Instead, the new arrangement will spread the errors in the frame, so it becomes easier to the Viterbi decoder to correct these errors when the reception of the frame is performed.

Once again, the modulation chosen for the established communication determines which permutations shall be applied to the sequence.

The size of each interleaving block corresponds to the number of bits in a single OFDM symbol, N_{CBPS}, which means that the modulation used in a transmission determines the number of bits considered as a group by the interleaver.

Regarding the four modulations of an OFDM system, the size of each interleaving block is represented in the next table:

Modulation	N_{CBPS}
BPSK	48
QPSK	96
16-QAM	192
64-QAM	288

Table 1. Interleaving block size.

Both the interleaver and the deinterleaver receive from the management blocks mentioned in the previous sections the information of the modulation used, so they can perform the appropriate permutation and interleaving (or deinterleaving) operation.

3. FPGA's usage and relevancy

In this project we are using a development board from Altera (the EP2S60F1020C4 device) which includes a Stratix-II FPGA[2].

The design and synthesis of the whole system was aided by Altera's software named *Quartus-II*. All the blocks were described using VHDL. The schematic form was used to assure the communication between blocks.

After its development, the physical layer was integrated in the whole system. The compilation of the produced system led to the following usage of the FPGA resources:

	Usage	Resources	Perc.(%)
ALUs	41204	48352	85 %
Pins	104	719	14 %
Memory Bits	1,065,600	2,544,192	42%
PLLs	1	12	8 %

Table 2. FPGA usage.

We must stress that, in the same device, both transmission and receiving chains where implemented. Moreover, the numbers above include a microprocessor for the Medium Access Control (MAC) layer, MAC/PHY interface blocks and lower PHY intermediate frequency (IF) stages including signal conditioning, transmission, reception, data and clock recovery.

The whole system was tested on a Portuguese road and a motorway using two units: one in the infrastructure and the other as an on-board unit of the vehicle.

These two units were identical and composed of:

- an Altera development kit EP2S60F1020C4;
- a pair of 5.9GHz antennae;
- a 5.9 GHz power amplifier;
- a transceiver;
- a car-PC.

The three use-cases under test were successfully demonstrated.

The use of a FPGA has shown to be extremely relevant (almost mandatory) due to the level of complexity of our system. In fact, the development of each block using physical components would be a time-consuming approach, regarding the fact that each change would lead to new a hardware circuit. If we think of the number of blocks that compose the whole system, we will certainly realize that such approach can not even be considered. One should note as well that such approach would lead to a bigger and more expensive solution.

Due to the fact that our communication system is still under development, the new changes can be easily implemented and tested using a FPGA, which would definitely not occur if a variety of components was used instead.

Moreover, there are many operations that must be done simultaneously in real time with tight intervals which demands a hardware approach rather than a software one.

4. Conclusions and future work

First of all, one should note that the use of FPGA has shown to be extremely relevant, almost mandatory, due to the complexity of our communication system.

The use of VHDL to describe blocks has proved to be successful.

Our prototype of a vehicular communication system is based on the IEEE Std 802.11p, considering the specifications of this standard.

The use of the resources of the FPGA device has margin for improvement and this issue is being addressed.

Finally, one should note that this prototype is still under development and it will keep being tested and refined in a near future.

References

[1] IEEE Std 802.11-2007, "IEEE Standard for Information Technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements", Part 11: Wireless LAN Medium Access Control and Physical Layer Specifications, pp.591-636, Jun. 2007.

[2] Altera Corporation, "Stratix II EP2S60 DSP Development Board", *Data Sheet*, May. 2005.

Architectural Solutions for Server Scheduling Communication within Ethernet Switches

R. Santos, A. Vieira, R. Marau, P. Pedreiras, A. Oliveira
DETI / IEETA
Universidade de Aveiro, Portugal
{rsantos, alexandrevieira, marau, pbrp, arnaldo}@ua.pt

Luis Almeida
IEETA - DEEC / University of Porto
4200-465 Porto, Portugal
lda@fe.up.pt

Abstract

The information exchanged in Network Embedded Systems (NES) is steadily increasing both in terms of quantity, size and complexity. For instance, applications comprising data originated in simple 10 bit ADCs side-by-side with multi-kilobyte variable bit-rate multimedia traffic are, nowadays, becoming a commonplace. Moreover, many NES are frequently subject to real-time constraints and thus the associated information exchanges are subject to timeliness requirements. However, existing real-time Ethernet protocols have difficulties in handling these streams efficiently, particularly in what regards the arbitrary arrival patterns and different QoS requirements.

To overcome these limitations, the authors proposed recently the integration of server-based traffic scheduling concepts within a customizable Ethernet switch, called FTT-enabled switch. The server scheduling unit can be placed in different points of the FTT-enabled switch architecture. The particular placement chosen has a noticeable impact in terms of server responsiveness, flexibility, hardware complexity and global system schedulability.

This paper presents a qualitative comparison about the different architectural solutions and presents a prototype implementation of the hardware-based architecture. Extensive experimental results are also included, showing the correctness of the server operation both in terms of bandwidth guarantees, traffic isolation and latency bounds.

1. Introduction

Switched Ethernet architectures present attractive features such as large bandwidth, cheap network controllers, high availability, easy integration with Internet and a clear path of evolution. These features are fostering the expansion of switched Ethernet architectures to new application areas such as high-speed servoing, target tracking in military systems or even the control of electrical protection systems in substations. However, COTS Ethernet switches are not designed to support the timeliness and safety requirements found in many of the application areas aforementioned due to aspects like blocking caused by long non-preemptive frames, lack of protection against errors in time domain, a limited number of priorities and possible memory overflows.

To address these limitations, diverse Real-Time Ethernet (RTE) protocols have been developed (e.g. [1], [2], [3], [4], [5], [6], [7]). However, most of the RTE protocols share a common difficulty in efficiently handling together real-time messages with diverse arrival patterns, such as periodic and aperiodic, treating them in different ways, frequently with static resource allocation for each case.

Server-oriented architectures are recognized as an effective means to enable such kind of communication resource sharing [8]. The current support for network partitions suffers from limitations imposed by specific medium access control and queues management policies within network controllers, network devices and protocol stacks that do not allow efficient server-based scheduling policies as those developed for CPU scheduling. Moreover, network partitions are typically static, as in TDMA-based approaches, and do not adapt to variations in number of active components in the system or in their requirements. Additionally, the respect for network partitions is frequently delegated to the end nodes that must execute a specific layer on top of the general network interface, typically a traffic shaper, which is a limitation for the integration of legacy systems and other general purpose systems that do not originally include such layer.

To overcome the limitations mentioned above, the authors proposed previously the Server-SE protocol [9], integrating the FTT-SE [2] and Server-CAN [10] protocols, the former providing a master/slave architecture that supports operational flexibility and the latter providing an integrated server-based traffic scheduling paradigm. Server-SE provides a seamless integration of real-time and non-real-time services, with strict timeliness guarantees to the first class. Arbitrary server scheduling policies are supported including their hierarchical composition. Furthermore, the servers properties can be changed dynamically, e.g., to deal with changes in the application requirements or environment, without compromising the timeliness of the real-time services. The FTT-SE framework was complemented recently with a customized Ethernet switch [11] that integrates the FTT master functionality and is capable of traffic classification and policing at the input ports. This latter feature allows confining the incoming traffic to reconfigurable time windows, whichever its type and arrival pattern. This capability is not present in current real-time Ethernet (RTE) protocols and is particularly well suited for supporting open distributed real-time systems.

The architecture of the FTT-enabled switch permits placing the server scheduling unit in different places. More specifically, the server scheduling can be carried out either in software, under control of the FTT Master module, or in hardware, operating complementary to the FTT Master module. This design option has important consequences in terms of responsiveness, flexibility, hardware complexity and global system schedulability. This paper presents the both architectural solutions, performs a qualitative comparison of them regarding the merit figures before mentioned and presents a prototype implementation of the hardware-based approach. Extensive experimental results are also included, showing the correctness of the server operation both in terms of latency bounds, bandwidth guarantees, traffic isolation and hierarchical server composition.

The remaining of the paper is organized as follows: Section 2 presents a brief overview on the related work; Section 3 presents a brief overview about server-based traffic scheduling; Section 4 describes the implementation of software and the hardware-based architecture and discusses their advantages and disadvantages; Section 5 presents experimental results on the hardware implementation and, finally, Section 6 presents the conclusions.

2. Related Work

The nomenclature given to servers in the networking domain is frequently different from the one used in CPU scheduling. For example, a common server used in networking is the *leaky bucket*. This is a specific kind of a general server category called *traffic shapers* [1], which purpose is to limit the amount of traffic that a node can submit to the network within a given time window, bounding the node burstiness. These servers use techniques similar to those used by CPU servers, based on capacity that is eventually replenished. Many different replenishment policies are also possible, being the periodic replenishment as with the Polling Server (PS) or the Deferrable Server (DS), the most common ones. However, it is hard to categorize these network servers similarly to the CPU servers because networks seldom use clear fixed or dynamic priority traffic management schemes. In fact, there is a large variability of Medium Access Control (MAC) protocols, some of them mixing different schemes such as round-robin scheduling, first-come-first-served, multiple priority queues, etc.

Focusing on RTE protocols, some limited forms of server-based traffic handling can be found. PROFINET RT and IRT [6] present bi-phase periodic communication cycles, comprising a mandatory Real-Time (RT) phase eventually followed by an optional non RT (NRT) phase. The RT schedule is built off-line and downloaded to the switch at configuration time. The protocol depends on a custom switch to enforce the cyclic structure and traffic confinement. The protocol operation can be regarded as a polling server, devoted to the periodic traffic, composed with a background server, dedicated to the NRT traffic. The TTEthernet [7] switch is also based in a customized switch that enforces a TDMA framework. When there is no RT traffic, nodes can transmit arbitrary NRT data. Whenever a TDMA slot is scheduled, the switch aborts current ongoing NRT transmissions, if any, making sure that the communication medium is free for the RT transfer. The underlying TDMA framework permits the existence of event slots thus, globally, the operation of this protocol can be regarded as a set of polling servers (off-line scheduled event slots) combined with a background server that handles the NRT traffic. Ethernet Powerlink [4]) also presents a TDMA scheme, based on a cyclic bi-phase communication structure, with one phase devoted to the isochronous traffic and the other to aperiodic traffic. The protocol operation can be regarded as the composition of two polling servers, one devoted to the isochronous traffic and the other to the asynchronous traffic. The protocol is based on a Master-Slave access control scheme, thus servers are scheduled in software. Other protocols, such as [1], implement traffic shapers in the end nodes, managed by suitable software modules, which behave similarly to a DS.

Due to infrastructural limitations, none of these protocols supports arbitrary server policies nor their hierarchical composition and dynamic adaptation or creation/removal, features that are provided by the Server-SE implementation described in this work.

3. FTT-Enabled Switch

The FTT-enabled switch is based on the Flexible Time-Triggered (FTT) paradigm with the FTT master included inside the switch (Master Module in Figure 1). The FTT protocol defines three traffic classes: 1) periodic real-time messages activated by the master (referred to as *synchronous* since their transmission is synchronized with the periodic traffic scheduler); 2) aperiodic or sporadic real-time traffic, autonomously activated by the application within each node, and 3) non real-time traffic. Classes 2 and 3 are referred to as *asynchronous*. The synchronous and asynchronous traffic are transmitted within separate windows with the former typically having priority over the latter. The non real-time traffic is scheduled in background, within the asynchronous window. For the synchronous traffic, a master/multi-slave transmission control technique is used, according to which a master addresses several slaves with a single poll message, considerably alleviating the protocol overhead when compared to the conventional master-slave techniques. The communication is organized in fixed duration slots called Elementary Cycles (ECs). Each EC starts with one poll message sent by the master, called Trigger Message (TM). The TM contains the schedule for that particular EC. Only the messages that fit within an EC are scheduled by the master, thus memory overflows inside the switch are completely avoided for such kind of traffic.

In short, the FTT-enabled Switch provides the following advantages: 1) Online admission control, dynamic quality-of-service management and arbitrary traffic scheduling policies; 2) an increase in the system integrity since unauthorized real-time transmissions can be readily blocked at the switch input ports, thus not interfering with the rest of the system; 3) the asynchronous traffic is autonomously

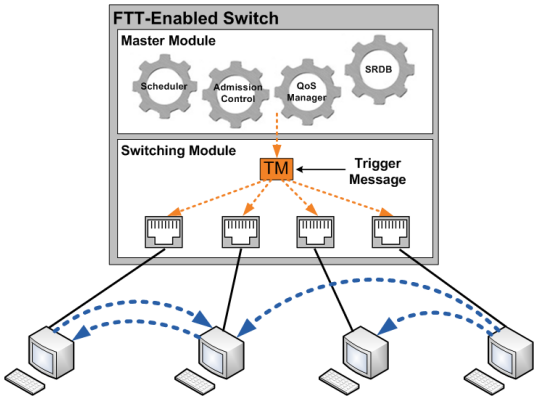


Figure 1. FTT-enabled Ethernet switch.

triggered by the nodes; 4) a seamless integration of standard non-FTT-compliant nodes without jeopardizing the real-time services.

4. Server scheduling integration analysis

The server scheduling in the FTT-enabled switch can be carried either in software, under control of the FTT Master module, or in hardware, operating complementary to the FTT Master module. This design option results in differentiated behaviors in terms of responsiveness, flexibility, hardware complexity and global system schedulability. This section explores both these architectural design options, showing its operation principles and presenting a qualitative comparison among them.

4.1. Servers implemented in software, inside the Master Module

Following a pure software-based approach, the server scheduling can be carried out at the Master node, whose arquitecture is represented in Figure 2. From the logical operation point of view, this approach is essentially equivalent to the Server-SE protocol [9]. The servers are software entities that reside in the master node. Each server has an associated memory block, organized as a FIFO. The traffic arrives via input ports and is submitted to the Classifier and Verifier Unit that classifies and validates the received messages. Whenever a valid message associated with a given server arrives, it is moved to the respective FIFO. Once every cycle the switch posts the Master about the status of the server FIFOs. Also once every EC the scheduler builds the EC-schedule and generates the trigger message, identifying the messages that should be transmitted in the following EC. The switch intercepts the EC-schedule and then forwards the messages associated with the scheduled servers. This scheme shares essentially the same properties as the Server-SE protocol, providing a great flexibility, permitting the support of arbitrary server schemes as well as its composition, combined with a tight integration with the Master scheduling, admission control and QoS management. However, as seen above, this approach still depends on a explicit signaling mechanism to post the scheduler about the occurrence of server requests. Additionally, the EC-schedule is built one EC in advance. As illustrated in Figure 3, this whole process results in a server latency between one and two ECs. Thus the server latency is strongly dependent on the EC duration and can be relatively large. This is the cost to pay for having the servers scheduling carried out by the master scheduler in a integrated fashion inside the Master Module.

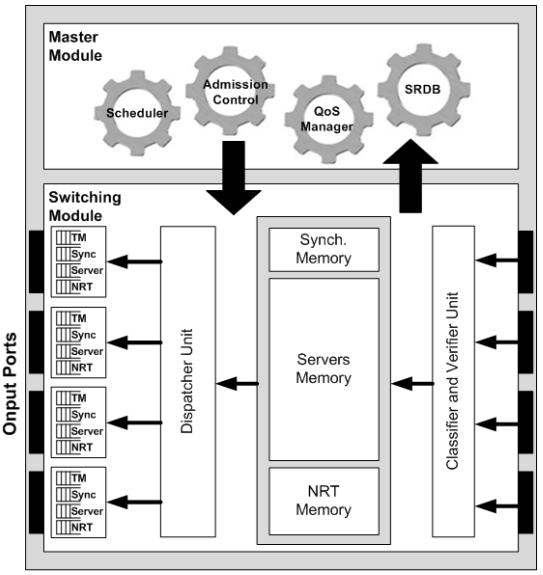


Figure 2. Functional Architecture.

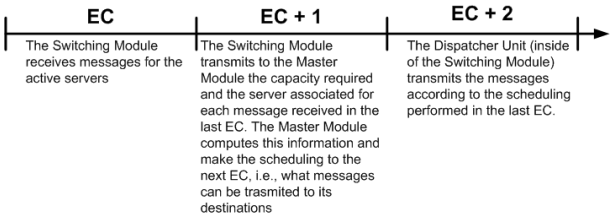


Figure 3. Servers forwarding process.

4.2. Server scheduling implemented inside the Switching Module

The server structures and their scheduling can also be implemented in hardware, inside the Switching Module. The servers are pre-configured and consequently their type and number cannot be changed online. A more dynamic architecture, permitting the dynamic creation and removal of servers, would require online FPGA reconfiguration, a subject that is outside of the scope of this paper. The admission of streams, namely the schedulability analysis and QoS negotiation, continues to be performed inside the Master Module. The negotiation procedure results are then intercepted by the Switching Module and used to configure the servers operational parameters.

Since the server scheduling is carried out independently of the master scheduler, it is necessary to break the EC in two sub-windows, one assigned to the master scheduler and

the other to the servers scheduler. Similarly to the software approach, whenever a valid message associated with a given server arrives, it is moved to the respective FIFO. Whenever the server sub-window is reached, the switch checks the server FIFOs, by priority order, and sends any pending messages until either the FIFOs become empty or the server sub-window finishes. This entire process is repeated every EC.

It is straightforward to conclude that, compared with the software architecture, this solution presents a greater reactivity. In the best case the latency is essentially the message transmission time, added with the switching latency, while in the worst case the message arrives at the end of the server sub-window, and thus has to wait to the beginning of the following server sub-window, which takes less than one EC time. Furthermore, for simple scheduling algorithms such as Rate Monotonic or Round-Robin, the implementation is resource-efficient and fairly simple. However, the hardware-based architecture compares negatively to the software one in terms of flexibility. On the one hand the number and type of servers is fixed, as mentioned above. On the other hand, complex servers can require a significant amount of hardware resources. Furthermore, from the global schedulability point of view, the hardware-supported servers also perform worse, since the master and the server scheduler are separate entities, unaware of the state of each other. For instance the master scheduler, when scheduling periodic messages, does not know the state of the servers. Thus, if the number of server requests is not sufficient to fill in the respective sub-window, the master scheduler is unable to reclaim that free space for scheduling periodic messages, thus penalizing the global system schedulability.

5. Experimental Results

The hardware-based server scheduling architecture was deployed in a prototype implementation of a 4 port FTT-enabled Ethernet switch architecture, following a similar Hw/Sw co-design approach as proposed in [12]. The prototype switch implements the Switching Module in hardware, using a NetFPGA board [13], integrating a Virtex-II Pro XC2VP50 FPGA and using 51% of the board FPGA total slices, with a maximum operation frequency of 126.20MHz. The Master Module is implemented in software, running in an independent CPU, connected to the FPGA by a dedicated Ethernet link on *Port 4*.

To assess the correct operation of the servers, it was created a configuration with an EC of 1ms, with the servers sub-window using 42% of the EC. Inside the server sub-window are created two sporadic servers, SS1 and SS2, with a budget of 3200B and period 1ms each. In addition it was also created a background server BS, to reclaim the bandwidth left by the sporadic servers. A video stream is simultaneously fed through servers SS1 and BS, while a time-bounded constant load, simulating an UDP transaction, is fed to SS2. The video stream has been analyzed offline, and offers an average load of around 10Mbps, with peaks that may reach 21.9Mbps. The load fed through SS2 is active from the instant t1=22 seconds to t2=58 seconds and, when active, generates a constant load of 48.9Mbps. A simple assessment of the load bandwidth submitted to the servers permits concluding that when the video streams experience peak activity the bandwidth is insufficient, leading to overloads. SS1 has the highest priority and thus the video stream served by it should not be degraded during overloads. The load traffic is served by a lower priority SS2. Since the bandwidth allocated to SS1 and SS2 exceed the server sub-window capacity, during peak activity on SS1, SS2 may also not be able to receive the full bandwidth. Finally, the video stream fed through the BS is expected to experience a severe quality degradation when the SS2 is active, since the BS has no guaranteed bandwidth.

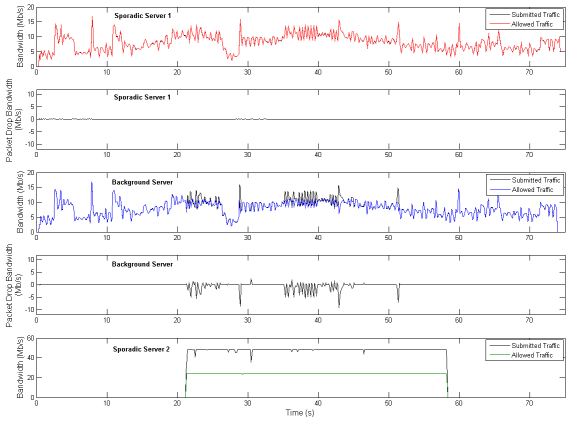


Figure 4. Submitted and forwarded load difference

Figure 4 shows, for each server, the instantaneous input and output traffic bandwidth. The first graph regards the highest priority server SS1 and, in this case, the input and output plots essentially overlap, meaning that the traffic managed by this server is forwarded without a noticeable delay. The second graph respects the video stream served by the BS. Between the instants t1 and t2, corresponding to the instants in which SS2 is active, it is possible to observe several sections in which the input and output traffic plots deviate from each other. The sections in which the input traffic plot is over the output traffic plot corresponds to periods in which the bandwidth allocated to the server is not enough to serve all the input traffic, potentially leading to packet losses. The switch has some buffer capacity, so in some sections the input plot is below the output plot, a situation that corresponds to the points in time where bandwidth allocated to the server is enough to reduce the amount of buffered messages. Finally, the last plot respects the UDP simulated load. As expected, the input and output traffic plots overlap most of the time, with occasional deviations coincident with peaks in the video stream served by SS1.

Figure 5 shows the input and output bandwidth of the three servers between t=32s and t=45s. Here is possible to observe the effect of the relative priority among servers. When SS1 has bandwidth peaks it is possible to observe a corresponding degradation on the BS, while the load traffic

978-972-789-304-1 REC'2010

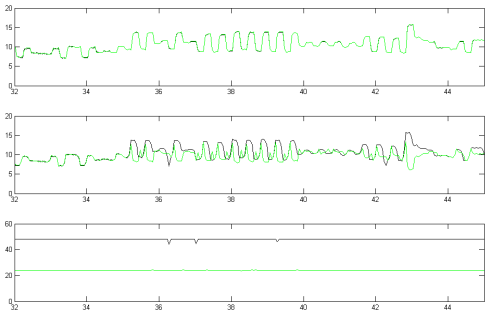


Figure 5. Submitted and forwarded load zoom

served by SS2 is essentially unaffected. This behavior is according with the expectations. SS1 was dimensioned to nearly fit the video stream peak bandwidth, and thus this stream is essentially unaffected by the switch. However, the bandwidth allocated to SS1 and SS2 exceeds the bandwidth of the server sub-window, so when SS1 uses the full bandwidth SS2 capacity is penalized. Finally, the BS receives the bandwidth left over by SS1 and SS2, thus being subject to an higher bandwidth degradation during peak activity of SS1 and SS2.

Table 1 depicts, for each server, the total number of packets transmitted and effectively forwarded by the switch during the experiment. The numeric results confirm the qualitative impressions above enunciated. The highest priority server SS1 experiences a marginal packet loss, while server SS2 experiences a slightly high packet loss ratio. This was expected due to the high bandwidth peaks of the video stream, leading to occasional situations in which the server sub-window capacity is exceeded. Finally, the BS experiences the worst packet loss ratio, as expected, since it is the lowest priority server, without any type of guarantees.

	SS1	**SS2**	**BS**
Packets submitted	54696	147492	54254
Packets forwarded	54642	73989	52679

Table 1. Number of packets submitted and allowed.

The switch latency was also assessed. For this purpose is was used a configuration previously described, A packet generator, served by the SS1, sent periodically 1500B packets to the switch. The packet generator period was set with a 0.4% offset relative to the EC length, to induce diverse phasing conditions with respect to the server sub-window occurrence. The switch was modified in order to return the packet to the sender, thus enabling the packet generator node to measure the round-trip delay. The obtained results are depicted in Table 2

The EC configuration used in this experiment is depicted in Figure 6. The best-case round trip delay happens when the packet transmitted by the generator to server SS1, arrives at the switch during the asynchronous window (server window). In this case it is forwarded immediately, suffering only a delay due to the packet transmission added by the processing overhead within the switch. The minimum measured delay was 125.8μs, 122μs due to transmission time (TT) and the remaining 3.8μs due to processing overhead within the switch (SD). On the other hand, the worst case situation occurs when the packet arrives at the switch and it is ready to be transmitted right after the end of the asynchronous window (at the beginning of the guarding window), being transmitted in the asynchronous window of the next EC. Therefore, in this case, the round-trip delay will be G_W + SW_W + TT (1500B) + SD = 140μs + 560μs + 122μs + 3.8μs = 825.8μs. The measured delay was 826μs, which is close of the expected value, showing the correctness of the implementation.

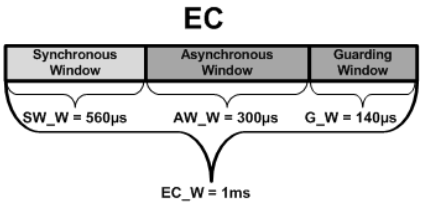


Figure 6. EC configuration.

6. Conclusions

Recently, the authors proposed an implementation of Server-SE over a new customized Ethernet switch that follows the FTT paradigm. The FTT-enabled switch supports a seamless integration of real-time and non-real-time services, copes with arbitrary traffic arrival patterns, allows arbitrary servers as well as their composition, and supports their dynamic creation and adaption. This paper presents preliminary work on the analysis of the different possibilities of implementation of the server scheduling, and associated tradeoffs, namely in what regards server responsiveness, flexibility, hardware complexity and global system schedulability. This paper also includes a prototype implementation of the hardware-based architecture and its experimental assessment. The experimental results show the feasibility and correctness of the implementation.

References

[1] Loeser, J. and Haertig, H. Low-Latency Hard Real-Time Communication over Switched Ethernet. In *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 13–22, Washington, DC, USA, 2004. IEEE Computer Society.

	Switch Latency
Minimum	125.9 μs
Maximum	826 μs
Average	371 μs

Table 2. Number of packets submitted and allowed.

[2] R. Marau, P. Pedreiras, and L. Almeida. Enhancing Real-Time Communication over COTS Ethernet Switches. In *WFCS 06 - The 6th IEEE Workshop on Factory Communication Systems*, Turin - Italy, June 2006. IEEE Computer Society.

[3] EtherCAT Technology Group. EtherCAT - Ethernet for Control Automation Technology. http://www.ethercat.org, December 2007.

[4] Ethernet Powerlink - online information. http://www.ethernet-powerlink.org/.

[5] Open DeviceNet Vendors Association. Ethernet/IP. http://www.odva.org/.

[6] PROFInet. Real-Time PROFInet IRT. http://www.profibus.com/pn, December 2007.

[7] TTTech. TTEthernet. http://www.tttech.com/solutions/ttethernet/, November 2008.

[8] Shin, Insik and Lee, Insup. Compositional real-time scheduling framework with periodic model. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–39, 2008.

[9] R. Marau, N. Figueiredo, R. Santos, P. Pedreiras, L. Almeida, and T. Nolte. Server-based Real-Time Communications on Switched Ethernet. In *CRTS 2008: First International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, Barcelona - Spain, 2008. .

[10] T. Nolte. *Share-Driven Scheduling of Embedded Networks*. PhD thesis, Department of Computer and Science and Electronics, Mälardalen University, Sweden, May 2006.

[11] R. Santos, R. Marau, A. Oliveira, P. Pedreiras, and L. Almeida. Designing a Costumized Ethernet Switch for Safe Hard Real-Time Communication. In *2008 IEEE International Workshop on Factory Communication Systems*, pages 169 – 177. IEEE Computer Society, May 2008.

[12] R. Santos, Vieira A. Marau, R., P. Pedreiras, A. Oliveira, and L. Almeida. A Synthesizable Ethernet Switch with Enhanced Real-Time Features. In *The 35th Annual Conference of the IEEE Industrial Electronics Society*. IEEE Computer Society, 2009.

[13] NetFPGA. http://www.netfpga.org/, May 2009.

Sessão Regular 7

Processamento de Áudio/Vídeo

Moderação: Paulo Flores
Instituto Superior Técnico / INESC-ID

Real-Time Stereo Image Matching on FPGA

Carlos Resende
DEEC, FEUP
ee04022@fe.up.pt

João C. Ferreira
INESC Porto, FEUP
jcf@fe.up.pt

Abstract

Real-time stereo image matching is an important computer vision task, with applications in robotics, driver assistance, surveillance and other domains. The paper describes the architecture and implementation of an FPGA-based stereo image processor that can produce 25 dense depth maps per second from pairs of 8-bit grayscale images. The system uses a modification of a previously-reported variable-window-size method to determine the best match for each image pixel. The adaptation is empirically shown to have negligible impact on the quality of the resulting depth map. The degree of parallelism of the implementation can be adapted to the available resources: increased parallelism enables the processing of larger images at the same frame rate (40 ms per image). The architecture exploits the memory resources available in modern platform FPGAs. Two prototype implementations have been produced and validated. The smaller one can handle pairs of images of size 208×480 *(on a Virtex-4 LX60 at 100 MHz); the larger one works for images of size* 640×480 *(on a Virtex-5 LX330 at 100 MHz). These results improve on previously-reported ASIC and FPGA-based designs.*

1. Introduction

Acquisition of three-dimensional information from images has important applications in computer vision [1] (including robotics [2], driver assistance [3] and surveillance [4]). This information can be obtained from stereo images in the form of dense disparity maps, which require the reliable establishment of correspondences between the images [5]. The computational effort of this task typically precludes achieving real-time performance with general-purpose processors. This has led to the development of various dedicated hardware systems [6, 7, 8, 9, 10].

A general approach to the calculation of the correspondences between the two images of a stereoscopic pair is based on a horizontal scan of the second image to find a matching position for each pixel of the first image. The matching pixel is the one whose neighborhood differs the least from the neighborhood of the pixel in the first image. Various metrics have been proposed [11], but the one based on the sum of absolute differences (SAD) of the neighborhood pixels is often chosen for hardware implementations due to its simplicity.

In the correspondence between stereo images using windows, the size of the neighborhood (size of the correspondence window) has a large influence on the quality of the matching. If the window is too small, the quantity of neighborhood information used is too small, producing errors of correspondence in large areas where pixel intensity is constant. On the other hand, if the window is too large, the quantity of neighborhood information used is too high, producing errors in the definition of object boundaries.

Since the quality of the matching depends so strongly on the correct size of the neighborhood, an adaptive window size should be used [12]. This approach has led to several implementations in dedicated hardware [13, 14, 8]. The more recent one [8] uses an Altera field-programmable gate array (FPGA) to process 64×64 pixel grayscale images well in excess of the target frame rate of 30 fps (frames per second).

We present a new FPGA-based implementation of the same general approach, that achieves a frame rate of 25 fps for grayscale images of 208×480 pixels (on a Virtex-4 FPGA) and 640×480 pixels (on a Virtex-5 FPGA). The algorithm used is a variant of the one employed in Ref. [8]. The 208×480 version has been integrated in a system that acquires images from two CMOS image sensors and displays the calculated disparity map on a VGA monitor in real-time.

The paper is organized as follows. Section 2 describes the correspondence algorithm used. Details of the hardware implementation are presented in Section 3. A second, expanded version of the hardware architecture with increased parallelism and capable of processing larger images was also developed, and is presented in Section 4. Section 5 analyzes the quality of the disparity maps obtained and the amount of resources used. Finally, Section 6 concludes the paper.

2. The Correspondence Algorithm

The system described here extracts three-dimensional information from images by calculating their disparity maps using a variant of the algorithm proposed by [8]. This modification reduces the quantity of neighborhood information used, and enables a simplified hardware architecture, with improved resource utilization and reduction of processing time.

The steps that constitute the modified algorithm are:

1. [Initialization] The algorithm starts with a window of size $w = 8$, as in the algorithm proposed by [8], where

it is stated that this value was empirically found to be the best starting window size. The algorithm divides the reference image in a grid and the candidate image in sections. The former constitute the various reference windows (RW in Figure 1), and the latter are the candidates considered during the search (CW in Figure 1).

Referring to Figure 1, the reference window represents the window (of the reference image) for which a correspondence is sought, the candidate windows are situated along a scan-line that covers the entire search area, and MW is the matching window, i.e., the candidate window with lowest SAD score.

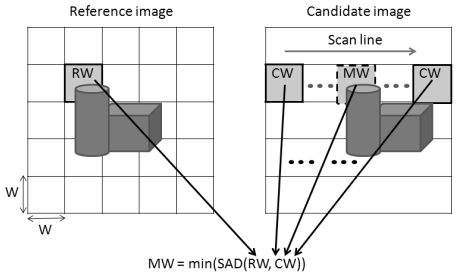


Figure 1. Set of reference and candidate windows (according to [8]). RW represents the reference window, CW the candidate windows, and MW the matching window with the lowest SAD (one of the CWs). The search for correspondence is made along the full scan-line.

2. [Select search area] Select the first section of eight lines.

   This step contains the largest difference between our algorithm and the version of [8]. While our algorithm applies the following steps to independent sections of eight lines, restricting the quantity of neighborhood information used to one section, the original one applies them to the entire image.

3. [Find best candidate] Calculate the matching between the reference window and all the candidates, by applying equation 1 to the various candidates windows and selecting the one that provides the lowest value (see also Figure 1).

$$\sum_{w}^{j=0}\sum_{w}^{i=0}|I_r(U_r+i,V_r+j)-I_c(U_c+i,V_c+j)| \quad (1)$$

   The functions $I_r(x,y)$ and $I_c(x,y)$ represent the intensity of pixels at position (x,y) in the reference and candidate images, respectively. Points (U_r,V_r) and (U_c,V_c) represent a reference pixel, which is used as the anchor point for the calculation of disparities between the reference and candidate images.

4. [Calculate disparity] Determine the disparity between the reference and the best candidate window from the previous step. Given the points corresponding to the best match (x_r,y_r) in the reference window and (x_m,y_m) in the candidate window, the disparity is given by $d=|x_r-x_m|$. (It is assumed throughout that $y_r=y_m$, i.e., the reference and candidate cameras are vertically aligned.)

   Disparity can be interpreted as the inverse of depth: pixels with larger disparities belong to objects that are nearer to the cameras.

5. [Shrink window] If $w \neq 1$, the window size is reduced by half horizontally and vertically.

   The situation of the reference and candidate windows is shown in Figure 2, where $w=8$ and the new windows RW (reference window), CW (candidate window) and MW (matching window) have an horizontal and vertical size of 4. The refined search for the matching window is restricted by considering the neighborhood information of the previous step.

   Figure 2 represents the following situation: the search of correspondence for windows with $w/2$ is restricted to the position where the two neighborhoods with $w=8$ have found the best correspondence (regions represented by shifts of $\pm d$ around the MW and CW windows, where $\pm d$ represents all the candidate windows of size 4 inside the matching window of the neighbors with size 8). In this way, the search is restricted to those 2 regions, because that is the maximum number of windows with size 8 for each window with $w=4$. This happens because the sections under consideration have a height of 8 lines, resulting in the existence of neighbors only on the left and right sides of each window. However, when $w=2$ and $w=1$, the number of neighbors of each window increases to 4 (neighbors on the left, right, above and below), and so does the amount of neighborhood information.

6. [Iterate] While $w \neq 1$, repeat from step 3.

7. [Proceed to next section] After calculating the correspondence for all pixels of a section, select the next one and repeat from step 3. If all sections have been processed, terminate.

3. System Architecture

The disparity processor implemented for this work is included in a system constituted by: a pair of CMOS cameras used to capture the images; a VGA monitor used to display the disparity maps and the reference and candidate images; and an evaluation board with the FPGA used to implement the processor and to establish the communication with the peripheral devices (CMOS camera and VGA monitor).

Image capture is done using two OV7620 CMOS cameras from OMNIVISION, which are controlled through an I2C interface. The evaluation board includes a Virtex-4 LX60 FPGA from Xilinx and all the peripherals used to communicate with the cameras and monitor. The interface

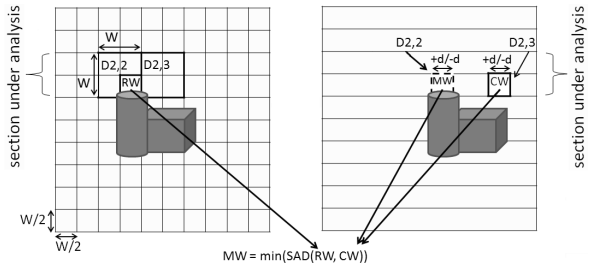


Figure 2. Set of reference and candidate windows of size $w/2$. The candidate windows analyzed at the lower window size must be inside the four regions that are within distance d from the position where the best correspondence for window size w was found.

with the VGA monitor, where the disparity maps are displayed, is made through an adapter card that takes care of all the synchronization necessary to correctly communicate with the VGA monitor.

3.1. Top-level Modules

The system implemented on FPGA is organized in the three top modules shown in Figure 3: a) data acquisition and control; b) SAD tree; c) calculation of correspondences. The last two modules together comprise the unit for the calculation of disparities. Depth map construction is done concurrently with image capture, and starts as soon as sufficient image data is available (one image section as described in Section 2).

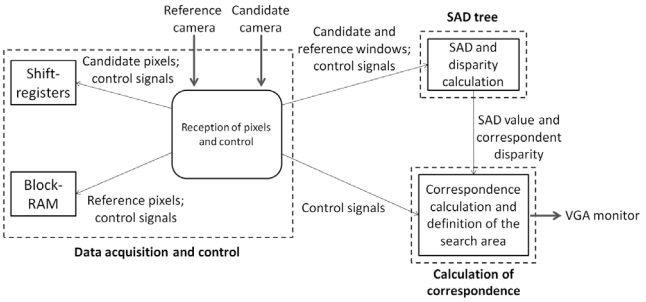


Figure 3. General view of the correspondence processor.

The module for data acquisition and control receives the pixels from the cameras and saves them in memory (shift-registers and Block-RAMs), controls of the size of the correspondence window and keeps track of its position in the image. This information allows the other modules to identify the current phase of the disparity calculation, and to update the control signals of their state machines accordingly.

The pixel intensity information and the control data concerning the windows being analyzed are sent to the module `SAD trees`, where the SAD metric (equation 1 of the correspondence algorithm) is applied to the multiple pairs of reference and candidate windows. Additionally, this module calculates the disparity associated with each one of these pairs.

The control data associated with the reference and candidate windows, and the associated disparity information are sent to the module for calculation of correspondences, which is responsible for defining the search area and for determining the best match amongst the candidate windows. The disparity for the matching window is stored in internal memory (in block RAM).

The disparity values calculated for the various windows are stored in block RAMs, whose depth and width depend on the associated window sizes. When the disparity values for all the pixels of a section have been calculated, they can be sent to the VGA monitor, while at the same time calculating and storing, in the same block RAMs, the intermediate disparities (disparities for windows of size 8, 4 and 2) of the next section. When the disparities for the windows of size one of the new section start to be calculated, the block RAMs used to store the disparities of windows of size one of the previous section are already free (because the disparities have already been sent to the VGA monitor), and can be used to store the new values.

Two different clock frequencies are used in the system: 12.5 MHz for the acquisition of pixel data from the CMOS cameras, and for sending the disparity information to the VGA interface; and 100 MHz for the core that processes the stereo images and determines the disparity information.

For the implementation of these modules various resources available on the FPGA are used: Block-RAM and shift-registers are employed for the memory structures used to save the pixels received from the cameras and the disparity information obtained for each window size; adders are used for the implementation of the SAD modules; and a DCM is used to generate the clock signals. A more extensive analysis of each of these units follows.

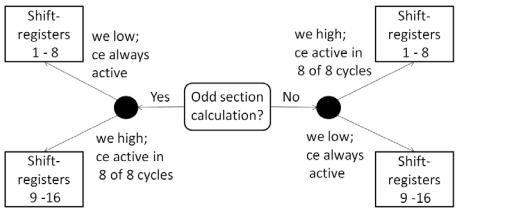
3.2. Management of Image Data

The image acquisition module uses two types of memory structures: shift registers for the pixels of the candidate image, and Block-RAM for the pixels of the reference image. This difference is justified by the different behavior of the two window types. Reference windows are shifted at least by eight positions and can, therefore, be efficiently implemented in Block-RAM. (The precise amount depends on the quantity of parallelism used: for the implementation being discussed they are shifted by 16 pixels, because the correspondence is made for two reference windows simultaneously). Candidate windows are shifted by one pixel, which is harder to implement in Block-RAM, but easily implemented by shift registers. This is another difference in comparison with the reference implementation, which uses shift-registers for both images, resulting in a significant increase in the number of logic blocks used.

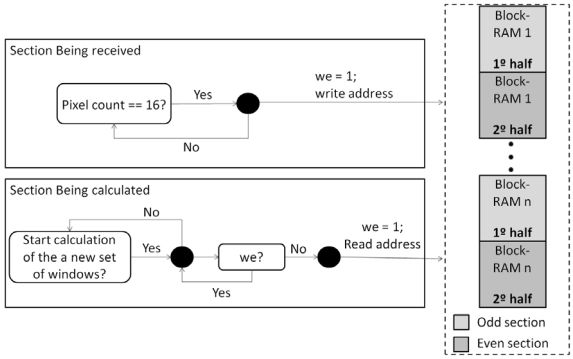
Although the pixel rate is 12.5 MHz for the two image sources, both memory structures operate at 100 MHz, since the memory units must provide image data at this rate to the disparity calculation modules.

In order to guarantee that the read and write accesses to

the memory modules are done without collisions, different approaches are used for the two types of memory structures, as shown in Figure 4. In this figure the CE and WE symbols represent the chip enable and write enable signals, respectively, and the word "section" always refers to the section of eight lines mentioned in the algorithm description (Section 2).



(a) Shift-registers access.



(b) Block-RAM access.

Figure 4. Coordinating access to image data.

The memory organization for candidate images uses two sets of shift registers: one set is used to store the section of eight image lines being analyzed at the moment, while another set is used to store the image lines being acquired at the same time. This is why the number of shift registers used in the implementation is 16. Figure 4(a) shows that the odd sections of eight lines are saved in the first eight shift-registers, and the even sections in the other eight. The depth of the shift registers is equal to the width of the images.

Figure 4(a) shows that, for write operations, each shift register is only active every eighth cycle of the 100 MHz clock. Read operations are done on every cycle, so that the pixels of the new candidate windows are sent to the SAD tree at the correct rate (100 MHz).

For the block-RAM-based reference window, the data for the section being currently processed and the section being acquired share the same physical memory, so access synchronization is more elaborate. As can be seen in Figure 4(b) each Block-RAM are divided in two halves: one half is used to save the pixels of the section being analyzed at each moment and the other half is used to save the pixels of the section being received. The first half is used to save the pixels of the odd sections of eight lines and the other half to save the pixels of the even sections.

Read access is only permitted when no write signal is active, as shown in Figure 4(b). This is done without delaying the calculation of disparities, since the write signal is only active once every 16 cycles of the 12.5 MHz clock. For each write operation, 16 pixels are committed to one memory position (Figure 4(b)). Thus, each memory position will contain all the pixels of a line of two reference windows.

The number of single-port block RAMs used for this approach (parameter n in Figure 4(b)) depends on the amount of parallelism used in the calculation of correspondences. In each cycle of the 100 MHz clock, a number of pixels equal to $8 \times 8 \times p$ (where p is the amount of parallelism supported) must be read from memory. Since the width of each block RAM is limited and it can only be accessed one position at a time (two, in the case of a dual port block RAM), it is necessary to use several block RAMs in parallel, so that a single read access provides the number of pixels required to exploit a processing core with parallelism of order p.

3.3. Calculation of Disparities

The unit responsible for the calculation of disparities is organized in two levels (see Figure 5). The first level contains the modules that calculate the SAD values (using a WPPP architecture with parallel processing of both reference and candidate windows [8]) and the corresponding disparity. The number n of SAD trees used determines the amount of parallelism used in the implementation. The second level determines the search area and calculates the correspondence for each reference window.

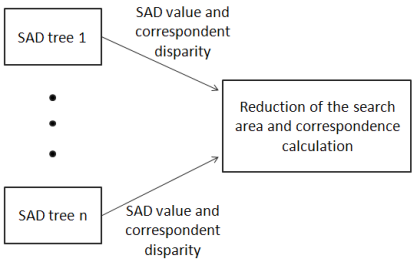
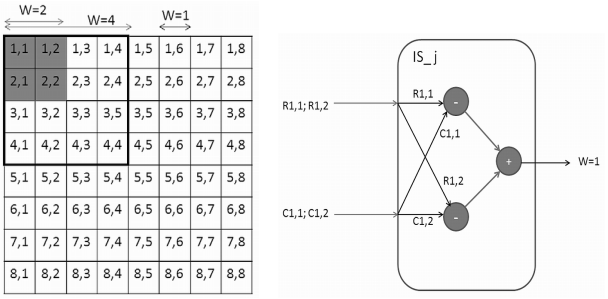


Figure 5. Calculation of disparities.

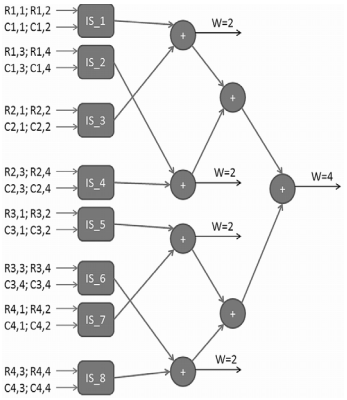
Figure 6(c) shows the constitution of each SAD tree: $R_{i,j}$ and $C_{i,j}$ represent the intensity of pixel (i, j) from the reference and candidate windows, respectively; the block IS_j is the element that calculates SADs for windows of size one (the absolute difference of two pixel values), as shown in more detail in Figure 6(b); and the rest of the SAD tree is composed of adders that combine the various absolute differences according to the window size. The example in the figure has an initial window size of four, but the analysis is valid for any size that is a power of two.

Figure 6(a)(a) represents a general correspondence window (candidate or reference). Establishing a correspondence between this window and the SAD tree of Figure 6(c) it can be seen that, for the different sizes of their sub-windows ($w = 2$ for sub-windows of size two, $w = 1$ for sub-windows of size one) the pixels considered in the calculations correspond to the pixels constituting each sub-window. This happens because the operations on SAD calculation are only additions and subtractions, allowing their

ordering on the SAD tree. With this, the pixels at each IS_j block are the same independently of the window size.



(a) Correspondence window. (b) Initial state.



(c) SAD tree.

Figure 6. General architecture used for the calculation of SAD.

This direct connections between memory modules and SAD units solve one of the major problems of this kind of implementations, that is, the need for multiplexers between memory modules and the processing units to allow a correct analysis of pixels when the window size is reduced.

Due to the block-RAM-based memory access scheme, disparity values are not obtained at a constant rate, since the speed of calculation depends on the number of cycles spent waiting until a read access is granted. In this case, the various registers keep their values until the pixels of the new window are read and the calculation is restarted. However, despite the variable data rate, a frame rate of 25 frames/second can be guaranteed.

The constant frame rate of 25 frames/second is guaranteed by the maximum processing time, which can obtained from the following expression:

$$T_{proc} = T_{store} + L_i + T_{calc} + L_f, \qquad (2)$$

where:

1. T_{proc}

   is the processing time (in seconds).

2. $T_{store} = \frac{208\times 8}{12.5\times 10^6}$

   is the storage time for the pixels of a new section: 208×8 is the size of each section and 12.5×10^6 is the rate at which the pixels are received (in hertz).

3. $L_i = \frac{7+8+3}{10^8}$

   is the latency before the calculation. The first seven cycles of latency are due to the fact that pixels are received at a rate of 12.5 MHz, and the signal indicating that all data has been received is only updated seven cycles of the 100 MHz clock after the reception of the last pixel. The following eight cycles of latency represent the number of cycles required to store the first pixel of the new section. The final three cycles of latency are the time required to retrieve data from the block RAM and to update the synchronization signals for starting the the calculation of disparities.

4. T_{calc} is the time taken by the disparity calculation (in seconds). Due to the block RAM memory access scheme this value is variable:

   (a)

   $$\frac{208\times\frac{208}{16}\times 4+\frac{208}{16}\times 1\times 4}{10^8}$$

   is the minimum time necessary for calculating disparities. $208\times\frac{208}{16}\times 4$ is the number of shifts required to cover the section in question, for all window sizes; $\frac{208}{16}\times 1\times 4$ is the minimum number of clock cycles (100 MHz clock) for the memory access that reads the pixels of a new window (for a complete section).

   (b)

   $$\frac{208\times\frac{208}{16}\times 4+\frac{208}{16}\times 8\times 4}{10^8}$$

   is the maximum time required for calculating disparities. $208\times\frac{208}{16}\times 4$ is the number of shifts required to cover the section in question, for all window sizes; $\frac{208}{16}\times 8\times 4$ is the maximum number of clock cycles (100 MHz clock) that may be necessary to read the pixels of a new window due to contention during block RAM access.

5. $L_f = \frac{4+2}{10^8}$

   is the latency after the beginning of calculation. The first four cycles of latency come from the delay between the beginning of the displacement of candidate windows and the return of the first SAD. The following two cycles of latency represents the number of clock cycles between the return of the first SAD value and the return of their disparity information (100 MHz clock).

This results in a minimum processing time of 0.242 ms and a maximum of 0.246 ms. This processing time, together with the frame rate restriction of the external hardware (CMOS cameras and VGA monitor) allows, as already stated, an image processing rate of 25 frames per second.

4. Expansion of the Architecture

The size of the image processed is highly dependent on the amount of resources available to implement the architecture presented in section 3.

There are three units that may limit the size of the images processed, due to lack of FPGA resources. They are: the shift-registers used to store the pixels of the candidate images; the adders used to implement the SAD tree; and the logic path (number of slices) used to define the search area. The last two units are fundamental to implement the parallelism necessary to satisfy the real-time requirements of the task.

Thus, to increase the image dimensions it is necessary to have enough resources to: increase the depth of the shift-registers used to save the pixels from the candidate image; increase the quantity of parallelism used to calculate the disparities; and increase the number of block RAMs used to store the pixels from the reference image and the disparities calculated for the various window sizes. Although the block RAMs are a fundamental unit of the correspondence processor, they do not represent a limitation in the hardware platform used, since their occupation is below 50%, as can be seen on table 2.

An expansion of the architecture from the previous section was implemented in a Virtex-5 LX330. The new version is capable of handling images of size 640×480. The quantity of parallelism necessary to cope with the larger image size, while keeping the frame rate of 25 frames/second was obtained from the following expression:

$$T_A = \frac{640 \times \frac{640}{Q_{par}} \times 4 + \frac{640}{Q_{par}} \times 8 \times 4}{10^8} = \frac{640 \times 8}{12.5 \times 10^6}$$

where:

1. $T_A = \frac{640 \times 8}{12.5 \times 10^6}$ is the storage time for the pixels of one section;
2. $\frac{640 \times \frac{640}{Q_{par}} \times 4 + \frac{640}{Q_{par}} \times 8 \times 4}{10^8}$ is the maximum time for disparity calculation;
3. Q_{par} is the amount of parallelism, i.e., the width of pixels analyzed concurrently. For example, with a parallelism of 2 windows, the width is $8 \times 2 = 16$ pixels.

Therefore, the amount of parallelism that must be supported is $Q_{par} = 40.5$. This corresponds to $\frac{40.5}{8} = 5.06 \rightarrow 6$ windows analyzed in parallel.

Although a minimum of six windows is required, the expanded architecture parallel uses eight. The reason is that six is not a divisor of 640, so it would be necessary to introduce additional circuitry to control the displacement of the reference windows, making the implementation more complicated.

Although the expanded architecture has been validated for images with 640×480 pixels, it is able to process 1016-pixel wide images. Only the depth of the shift-registers and block RAMs needs to be increased appropriately (which is feasible for the Virtex-5 LX330).

5. Results

5.1. Disparity maps

Using a simplified version of the reference algorithm, as discussed in section 2, does not result in a serious impact on the disparity map obtained. The confirmation of this result was done by comparing each pixel of the disparity map obtained by the reference algorithm with the corresponding pixel of the disparity map obtained by the simplified algorithm. This evaluation was made using images from the database presented in [15].

For both processors, the one implemented in the Virtex-4 LX60 and the one implemented in Virtex-5 LX330, it was necessary to cut the images available to the size processed by each implementation. Since the images of the database are in color, we converted the original images from the PNG (Portable Network Graphics) format to the PGM (portable gray map) format, which is easier to use.

We compared the results of our implementation of the reference algorithm in Matlab with the outputs of the Verilog description of the matching processor as executed on the Modelsim simulator. The disparity maps obtained from the Verilog version were compared pixel by pixel with the disparity maps produced by the Matlab version.

Results are shown in Figure 7 and in Table 1. The comparison does not include those pixels of the right region of the reference image that are not presented in the candidate image, since they do not have "correct" disparity values in either case (since no corresponding object actually exists). Although the test was done for the Virtex-4 LX60 and Virtex-5 LX330 implementations, only the results for the Virtex-4 LX60 study are shown, since the others are similar.

Table 1. Mean of the absolute differences of disparities.

	Mean (pixel distance)
Test image 1	3.201
Test image 2	2.49
Test image 3	1.343
Test image 4	1.16

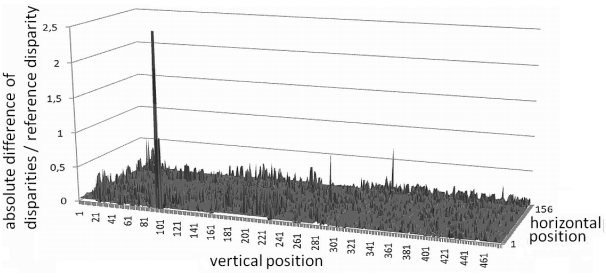


Figure 7. Relative difference of disparities for test image 4. The results for other images are similar.

Table 1 shows that for four different images the mean

978-972-789-304-1 REC'2010

absolute difference of disparities is very low, with a maximum value of three. "Pixel distance" is the difference of pixel position determined by the implemented algorithm and the one found by the original algorithm.

For most pixels ($\geq$90%) the absolute difference of disparities is less or equal to seven. Figure 7 represents the relative error (difference between hardware disparity and reference disparity as obtained by the original algorithm, divided by the reference disparity) for test image 4. The axes of Figure 7 labeled "vertical position" and "horizontal position" represent the vertical and horizontal position of each pixel. Similar results were obtained for all other tests. Figure 7 shows that the ratio is always near to zero, except for some sporadic peaks (the major one reaches a value of 2.3), which are due to occlusion. Because of that phenomenon, the disparity values for those regions are considerably different in both versions, since they react differently in this case.

The occlusion phenomenon mentioned in the previous paragraph occurs because of two reasons: some objects are not represented in the candidate image, since they are occluded by an object closer to the cameras; objects located at the right limit of the reference window are not represented in the candidate image.

5.2. Resource Utilization

Table 2 summarizes the utilization of FPGA resources for the two versions of the image matching processor. Comparison of the slice utilization in the two processors cannot be made directly, since slices in the Virtex-4 architecture (two flip-flops and two 4-input look-up tables) are different from slices in the Virtex-5 architecture (four flip-flops and four 6-input look-up tables). The high-level synthesis process was oriented towards maximizing clock frequency at the expense of FPGA occupation.

Table 2. Resource utilization for baseline and expanded processors.

Resources	Utilization (number)	Utilization (%)
Virtex-4 LX60 (208×480 pixels)		
Slices	20101	75
Block-RAM	64	40
Virtex-5 LX330 (640×480 pixels)		
Slices	50340	24
Block-RAM	168	58

Analyzing the occupation of the Virtex-4 LX60, the high utilization of slices is mainly due to the shift registers used for storage of two sections of 208×8 pixels of the candidate image, and to the quantity of parallelism used in the calculation of the disparities. The quantity of block RAM used is due to the storage of the reference image and the disparity values at each window size (for windows of size one, two, four and eight).

For the larger implementation on the Virtex-5 LX330, the number of slices used is determined by the shift registers, but also by the amount of parallelism used in this implementation, which is 4 times higher than the implementation on the Virtex-4 LX60. The number of block RAMs increases greatly for the same reason. This happens because the only way to increase the quantity of information that is available at each instant is to instantiate more block RAMs.

We can be concluded that the resource utilization depends on the size of the images analyzed and on the frame rate required. Images with larges dimensions require deeper shift-registers, in order to store the sections of eight lines of the candidate image. Relatively to the frame rate, higher frame rates require higher parallelism, which implies: the use of more block RAMs, needed to save all the reference windows analyzed at each instant; and the increase of slice utilization to implement the SAD and all the logic necessary to achieve the required parallelism.

The comparison between the implemented processor and the reference implementation, relatively to resources utilization, needs to be done carefully, since the types of FPGA used are different. However, the reference implementation [8] presents higher resource utilization, since it uses 42,508 logic elements of an APEX20KE from Altera (each consisting of a 4-input look-up table and one flip-flop), while the smaller of our implementations uses 19,978 slices of a Virtex-4 (two 4-input look-up tables and two flip-flops), for a total of 31,880 look-up tables and 16,951 flip-flops).

The lower resource usage of the proposed architecture is due mainly to the reduction of neighborhood information processed, and to the use of block RAM for storing the pixels of the reference image (instead of shift-registers).

5.3. Comparison with Other Approaches

This subsection presents a comparison between the implementation described in this paper with previously reported results, relatively to the dimensions of the images analyzed and the velocity of the processor. Table 3 summarizes the data. Column "time" represents the time spent to process one frame. For both implementations proposed in this paper, the one on Virtex-4 (Impl. 1 (V4)) and the one on Virtex-5 (Impl. 2 (V5)), the processing time is 40 ms, since both have a frame rate of 25 frames per second ($\frac{1}{25} = 40$ ms). Both are faster than the previously reported ASIC implementations [13, 14], but support a smaller maximum window size. Comparing with the FPGA implementation of Ref. [8], our implementations are able to process much larger images, while still satisfying real-time requirements.

6. Conclusion

This paper describes a hardware architecture for the calculation of dense depth maps from a pair of stereo images. The architecture is based on a modification of a previously reported variable-window-size method. Empirical tests indicate that the simplification introduced does not degrade

Table 3. Comparison between the proposed implementations and previous systems reported in the literature.

System	Image size	Max. window size	Freq. (MHz)	Time (ms)
Ref.[13]	512×512	25×25	200	60
Ref.[14]	320×240	15×15	125	100
Ref.[8]	64×64	8×8	86	0.19
Impl. 1 (V4)	208×480	8×8	100 MHz	40
Impl. 2 (V5)	640×480	8×8	100 MHz	40

The first two systems are implemented with CMOS ASICs: 0.5 µm and 0.18 µm technologies, respectively. Implementation [8] uses an FPGA from Altera (APEX20KE). The last two lines summarize the implementations described in this paper. All systems process 8-bit grayscale images.

the quality of the resulting depth maps. The proposed architecture admits implementations with a variable degree of parallelism, depending on the resources available. The architecture exploits the resources of modern platform FPGAs. In particular, the management of image data uses different memory resources for the reference and the candidate image, in order to take advantage of the different access patterns.

Two versions of the architecture with different resource requirements were implemented. Both produce dense depth maps in real-time (25 maps per second). The smaller implementations targets a Virtex-4 LX40 device and handles 208×480 images, while the larger one may use a Virtex-5 LV330 device (less than 60% of resource occupation) and handles 640×480 images. Additionally both are capable of finding a maximum disparity of 255.

Relatively to the velocity of the processor and the dimension of the images analyzed, the new implementations are faster than the previously reported ASIC implementations [13, 14], but support a smaller maximum window size. They are able to process much larger images than the FPGA implementation of Ref. [8], while still satisfying real-time requirements, as presented in Table 3.

References

[1] M. Z. Brown, D. Burschka and G. D. Hager, Advances in computational stereo, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol.25, no.8, pp. 993-1008, Aug. 2003.

[2] D. Murray and J.J. Little, Using Real-Time Stereo Vision for Mobile Robot Navigation, *Autonomous Robots*, vol. 8, Abr. 2000, pp. 161-171.

[3] U. Franke and S. Heinrich, Fast obstacle detection for urban traffic situations, *IEEE Transactions on Intelligent Transportation Systems*, vol.3, no.3 (2002), pp. 173-181, 2002.

[4] J. M. Manendez. L. Salgado, E. Rendon and N. Garcia, Motorway surveillance through stereo computer vision, *IEEE 33rd Annual 1999 International Carnahan Conference on Security Technology*, pp.197-202, 1999.

[5] D. Scharstein and R. Szeliski, A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms, *International Journal of Computer Vision*, vol. 47, Abr. 2002, pp. 7-42.

[6] M. Kuhn, S. Moser, O. Isler, F. Gurkaynak, A. Burg, N. Felber, H. Kaeslin and W. Fichtner, Efficient ASIC implementation of a real-time depth mapping stereo vision system, *Proceedings IEEE International Symposium on Micro-NanoMechatronics and Human Science*,vol. 3, 2003, pp. 1478-1481.

[7] J. Woodfill, G. Gordon and R. Buck, Tyzx DeepSea High Speed Stereo Vision System, *Computer Vision and Pattern Recognition Workshop CVPRW '04*, 2004, p. 41.

[8] M. Hariyama and Y. Kobayashi and H. Sasaki and M. Kameyama , FPGA implementation of a stereo matching processor based on window-parallel-and-pixel-parallel architecture, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* **E88-A** (2005) 3516–3522.

[9] S. Lee, J. Yi and J. Kim, Real-Time Stereo Vision on a Reconfigurable System, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2005, pp. 299-307.

[10] L. Mingxiang and J. Yunde, Stereo Vision System on Programmable Chip (SVSoC) for Small Robot Navigation, *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006, pp. 1359-1365.

[11] R. Porter and N. Bergmann, A generic implementation framework for FPGA based stereo matching, *Proceedings of the IEEE Region 10 Annual Conference on Speech and Image Technologies for Computing and Telecommunications*, vol. 2, 1997, pp. 461-464.

[12] T. Kanade and M. Okutomi, A stereo matching algorithm with an adaptive window: theory and experiment, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 16, 1994, pp. 920-932.

[13] M. Hariyama, T. Takeuchi and M. Kameyama, VLSI processor for reliable stereo matching based on adaptive window-size selection, *Proceedings IEEE International Conference on Robotics and Automation*, vol. 2, 2001, pp. 1168-1173.

[14] M. Hariyama and M. Kameyama, VLSI processor for reliable stereo matching based on window-parallel logic-in-memory architecture, *Digest of Technical Papers Symposium on VLSI Circuits*, 2004, pp. 166-169.

[15] D. Scharstein and R. Szeliski, Middlebury Stereo Vision, June 2009, `http://vision.middlebury.edu/stereo/data/`

Audio Mixture Digital Matrix
MIAUDIO

David Pedrosa Branco
Universidade de Aveiro
dpbranco@ua.pt

Iouliia Skliarova
Universidade de Aveiro
iouliia@ua.pt

José Neto Vieira
Universidade de Aveiro
jnvieira@ua.pt

Abstract

Modern music is turning more and more to technologic solutions so that new composition styles and techniques are created. Sound movement is a concept that is gaining strength in this area. Multichannel sound diffusion systems are built to provide the user with the capability to independently control several input channels through the desired output channels. This project (MIAUDIO) allows using up to 8 input channels that can be mixed in real-time through 32 output speakers. A hardware solution was adopted. Eight input analogue audio signals are conditioned, converted to digital format and sent to a Field Programmable Gate Array (FPGA). A host computer communicates with the FPGA via USB and supplies the parameters that define the audio mixture matrix. The FPGA processes this information and sends the resulting signals to digital-to-analogue converters so that the analogue signals are then filtered and reproduced. MIAUDIO was successfully implemented. This is a low-cost solution and its developing time was relatively short. A signal analysis has been made and good results have been achieved.

1. Introduction

Electroacoustic is turning more and more to sound diffusion techniques. With resource to new technologies multichannel sound systems are constructed. These systems allow creating different sound diffusion scenarios, i.e., immersion and the possibility of movement of the sound around the audience. SARC [1] and BEAST [3] [4] are some of multichannel sound diffusion systems. These systems use several loudspeakers that are strategically positioned around the audience. The most common disposition is known as the Main Eight concept [12]. In this speaker distribution, the listening room is divided in four sections: Main, Wide, Rear and Distant. The section Main gives us the frontal image while the Wide is used to stretch that image. The section Rear is positioned behind the audience allowing a 360° rotation of the sound. Finally, Distant, gives us the perception of what is further than the main image.

With resource to software tools, the mixture of the input channels is made and most of the times hardware is also used to define the mixing parameters. As we will see, these systems differ from the implementation topology used in MIAUDIO.

2. State of the Art

Sonic Arts Research Center

Sonic Arts Research Centre (SARC), located in Belfast, is a sound diffusion system that features 112 loudspeakers that reproduce the mixture of 24 audio input channels through 48 different outputs. The 112 loudspeakers are strategically installed along four levels. This sound diffusion system is controlled using three Digidesign 192 I/O audio interfaces [6] that interact with a Pro Tools HD3 Accel system. A personal computer, Apple PowerMac G5, runs the software (Pro Tools [7]) and, using the information provided by the Digidesign mixing surfaces, creates the mixture with the audio signals involved.

Brimingham Electroacoustic Theater

The Birmingham ElectroAcoustic Theater (BEAST) is another multichannel sound diffusion system. It was created in the Birmingham University in 1982. This system provides more than 100 speakers where each one can be independently addressed. Similarly to the SARC system, BEAST

uses a digital multichannel sound interface that is controlled via specially written applications using MIDI faders with resource to a software known as SuperCollider [2] [5] [13]. Using the software, the MIDI faders can be assigned so that they control a single, a pair or a set of speakers. This configuration offers good flexibility to this system.

Conclusions and Comparisons

Both systems presented use software based solutions. There is a software tool responsible for the mixture of the audio signals that uses information provided by digital mixture surfaces, or similar hardware. In this implementation method, a fast and reliable operating system is necessary so that real-time processing is guaranteed. The operating system has a great amount of resources dedicated to the sound system control leaving therefore little space to accomplish other possible tasks.

The project described in this article (MIAUDIO) has its mixing algorithm implemented in hardware. A Field Programmable Gate Array (FPGA) is used to receive the audio signals and process them according to the parameters that are sent by software. Being so, the software's responsibility is to send the information that defines the audio mixture – a task much simpler and less demanding than processing the mixture itself. This is one of the advantages in MIAUDIO. In software based solutions like in BEAST and SARC, the operating system that produces the mixture has to be extremely reliable and efficient but above all, has to have a great processing power. In MIAUDIO, given the simplicity of the task assigned to the operating system, there is space to introduce several new functionalities as masterization, sound effects, etc.

By adopting a hardware solution implementation, new functionalities can be introduced, in MIAUDIO, without changing the core of the system. Changes can be made at a higher level. It is possible to add software that interacts with the module responsible for sending the mixture parameters as well as to introduce additional hardware. The FPGA can also be reconfigured to add new features without having to change the rest of the hardware.

Another relevant fact in this project is related to its development time and cost. This project was developed in a relatively short amount of time when compared to similar systems. The cost of the components used to assemble the system is under 500 Euros.

3. MIAUDIO – Audio Mixture Digital Matrix

System Description

MIAUDIO is a multichannel sound diffusion system built around an FPGA of Spartan-3E family [14]. This system has the ability of mixing up to 8 analog input channels through 32 output channels. The analogue input audio signals are conditioned, converted to digital by several analogue-to-digital converters (ADC) and then sent to the FPGA that performs the mixing algorithm. The host computer connects to the FPGA and is responsible for sending the parameters that define the audio mixture, i.e., send the information that represents the intensity level of each input channel on each output. This topology can be interpreted as a matrix where each coefficient represents the level of each audio input on each output channel. Fig. 1 represents the system diagram. The host computer sends the parameters that define the audio mixture while the input channels, after the analogue-to-digital conversion, are sent to the FPGA. The resulting output channels are then converted to analogue so that they can be reproduced.

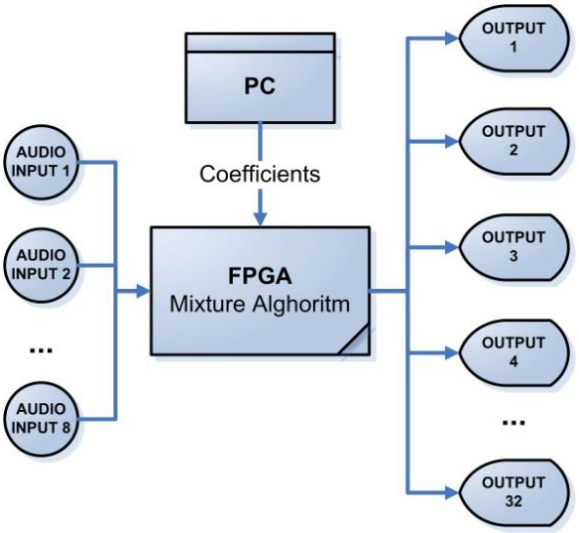


Fig. 1 - MIAUDIO's System Diagram

Internal Logic

Fig. 2 represents a block diagram of the several modules implemented in the FPGA. The Input block is in charge of the communication with the analogue-to-digital converters (ADC). After receiving a sample of each audio channel, this information is sent to the Arithmetic block whose

responsibility is to generate the 32 output signals according to the current mixture matrix. To obtain the parameters of the matrix, this block communicates with the Memory Control block that manages memory banks embedded in the FPGA where that information is stored. Because the matrix is controlled by a computer, the USB Communication block is created to establish the USB communication between the FPGA and the PC. After generating the 32 output samples, the Arithmetic block sends this information to the Output block that is responsible for properly sending these samples to the digital-to-analogue converters.

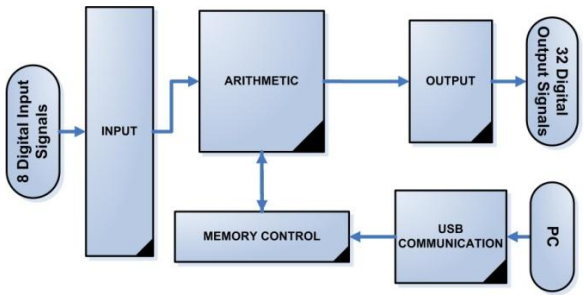


Fig. 2 - FPGA Internal Logic Blocks

Mixing Algorithm

Each audio input can have a different volume in each output channel. Being so, because there are 8 input channels and 32 outputs, 256 coefficients are necessary to define the audio mixture matrix. Each output can have information of any of the input channels, therefore each channel is multiplied by the coefficient that determines the weight of that input on the respective output and afterwards the 8 products associated with the same output are added. Fig. 3 represents the relation between the inputs, coefficients and outputs. As mentioned before, there are 256 coefficients that define the audio mixture matrix. Eight input signals are introduced in the system and 32 outputs are generated, being possible that each one of them is different combination of the input audio signals.

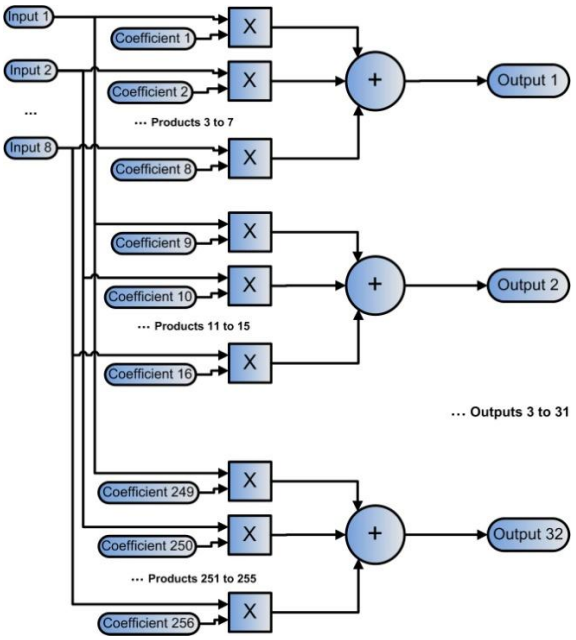


Fig. 3 – Arithmetic FPGA Logic

System Hardware

The system is built around an FPGA of Spartan-3E family [14]. To use this FPGA, the board NEXYS2 [11] from Digilent was chosen as the design platform. This board has numerous interfaces around the FPGA such as a USB module and several expansion ports that are directly connected to the FPGA. Considering that the analogue input signals are processed digitally, it is necessary to use analogue-to-digital converters (ADC) as well as digital-to-analogue converters (DAC). The converters selected for this project were PCM1802 ADC [10] and DAC8534 DAC [9], both designed by Burr-Brown Products. Additional hardware is also required to condition the signals to the system.

Fig. 4 represents the input and output stages, for two and four channels, respectively, of the system and their interconnections with the FPGA. The input signal is delivered through XLR [8] cables and introduced into input buffers that convert the signal from its differential format to single-ended. Then a second order antialiasing filter, implemented with resource to operational amplifiers, is used. The analogue-to-digital conversion is preformed and then the resulting information is sent to the FPGA. The input signal is converted with a 24-bit resolution and it is sent by the analogue-to-digital conversion through a serial interface. This transfer is controlled by the ADC. On the output stage, a similar but symmetric process occurs. The digital information is sent by the FPGA towards the DAC, also through a

serial interface. In this case, data has a 16-bit resolution. The analogue signal is low-pass filtered and then converted to differential format. To obtain the number of channels desired these blocks are replicated 4 and 8 times respectively.

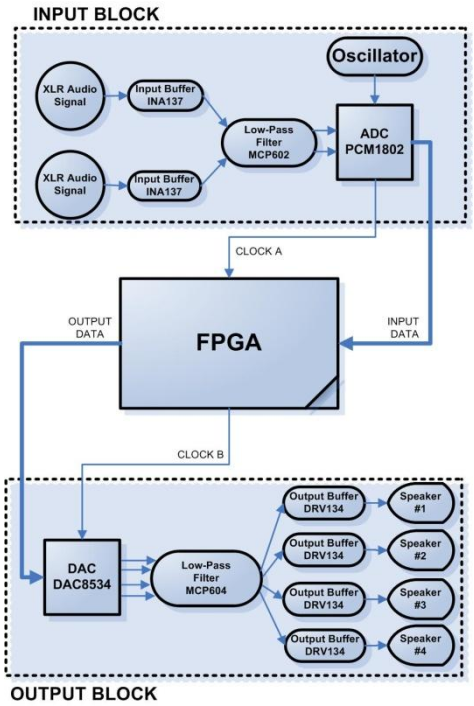


Fig. 4 - Input and Output Stages

Further Implementation Considerations

Evaluating the Arithmetic FPGA Logic, it is possible to observe that a total of 256 products are necessary once considered the products between each input signals and the respective 32 coefficients. Being so, it is crucial that this operation is optimized so that the processing time remains smaller than the ADCs sampling period. Therefore, dedicated multipliers where used to enhance the system's performance. A cyclic Finite State Machine (FSM) was created so that 16 of the 20 dedicated multipliers available in this FPGA where used in each loop iteration. Sixteen iterations are necessary to obtain the 256 products. To generate one output sample, 8 multiplications are necessary (each output is a combination of the 8 input signals). Being so, each iteration produces two output samples. A rounding algorithm and overflow detection is also accomplished while generating each output signal. Overflow detection is crucial because, after the described products, an eight operand addition takes place increasing therefore the probability of overflow occurrence. These algorithms will be briefly explained further in this article. The arithmetic block has all the data represented in two's complement format. The dedicated multipliers require this format as well as the analogue-to-digital converters.

Another crucial aspect is related to the clock synchronization. As we can see in Fig. 4, the ADC PCM1802 has a clock that controls the data transfer considering that it is configured in *Master* mode. Being this signal external to the FPGA (it is created by the ADC with resource to an external oscillator, and, in this case, has a frequency different from the 50MHz clock that controls the FPGA logic circuits), a *First-In-First-Out* (FIFO) stack was created. This FIFO is provided by Xilinx (*Xilinx LogiCORE™ IP*) and has the particularity of having, if desired, different write and read clocks. This module is highly effective and extracts possible synchronization concerns from the user. On the output stage, this issue is no longer a problem once the data transfer clock is generated by the FPGA. The digital-to-analogue converter works in *Slave* mode.

Fig. 5 represents the used rounding algorithm. This algorithm is applied after the addition operation is done. Considering that the data samples are, at this point, in two's complement format, to perform the rounding operation, it is necessary to evaluate the most significant bit. First, the less significant bit is evaluated. If it is equal to "0", no rounding is performed and these two bits are simply discarded. Otherwise, "1" is added if the most significant bit is "0" or is subtracted if the most significant bit is "1". After rounding, an overflow detection technique is necessary to confirm that no overflow has occurred.

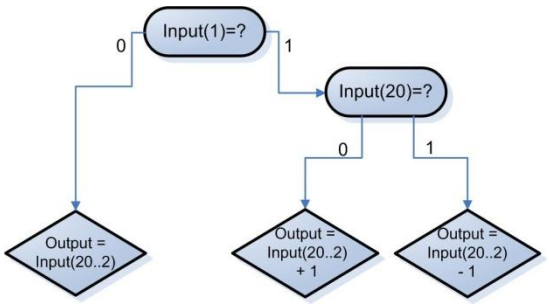


Fig. 5 - Rounding Algorithm

To allow overflow detection, an extra step was taken into account in the arithmetic addition phase. The most significant bit was replicated so that we would have four signal bits in the most significant data bits. This way, it is guaranteed that the resulting most

significant bit is intact after adding the eight inputs referenced to a certain output. Fig. 6 describes the implemented overflow detection.

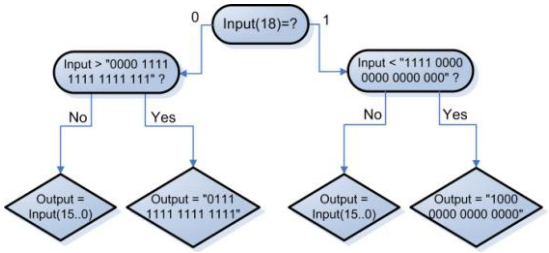


Fig. 6 - Overflow Detection Algorithm

By evaluating the most significant bit it is determined if the data is bigger or smaller than zero. If the most significant bit is "0", the word is compared to the greatest positive value possible, i.e., "0000 1111 1111 1111 111" in this example. If the most significant bit is "1" the data is compared with the greatest negative value possible, i.e., "1111 0000 0000 0000 000". When an anomaly is detected (data bigger than the maximum values) the data is assigned to the respective maximum value. We have therefore, saturated overflow detection.

Embedded in NEXYS2 there is a module responsible for managing the USB communication between the connected device and the FPGA. Cypress CY7C68013 [15] is an integrated circuit that interprets the USB communication signals and converts them to a sort of parallel communication. If the respective communication circuit (interacting with the Cypress module) is correctly implemented in the FPGA, the signals generated by the Cypress module are well interpreted and data can be transferred from a computer equipped with USB2.0 to the FPGA.

A source file that allows using this communication was provided by Digilent (manufacturer of NEXYS2) and adapted to this project. The adaptation consisted in storing the sent information in memory banks embedded in the FPGA. Previously, this information was stored in registers and there were only 16 register available. Considering that 256 registers would be necessary to store the matrix coefficients, it would be a waste of resources. While processing each group of 8 input samples, the memory banks are accessed so that the latest 256 coefficients are used.

4. Results

To evaluate the MIAUDIO's behavior, several tests were made during and after the final implementation. With the aid of a Logic Analyzer it was possible to determine the time interval between the beginning of the ADC's sample transfer and the instant where the DACs receive the corresponding sample. This time interval can be seen in Fig. 7 and corresponds to the FPGA processing time. It is equal to 13µs as shown in Table 2. Observing t_2 and t_3 duration, it is possible to verify that the sampling frequency is 96KHz. This matches the sampling frequency configured in the analogue-to-digital converters.

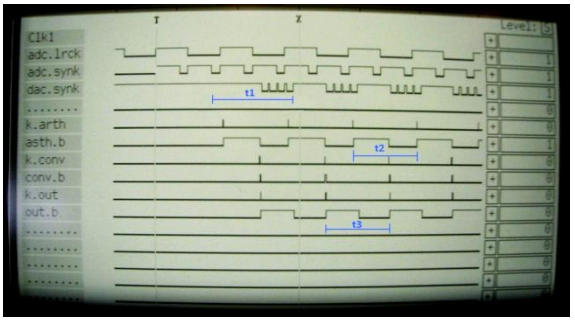


Fig. 7 - Test Time Diagram

Signal	Description
adc.lrck	Designates the channel being sent by the ADC (0 – channel 1 ; 1 – channel 2)
adc.synk	Represents the ADC data transmission state (1 – sending ; 0 – stopped)
dac.synk	Represents the DAC data transmission state (1 – stopped ; 0 – sending)
k.arth	Signals the beginning of the Arithmetic block processing
arth.b	Represents the Arithmetic block state (0 – standby ; 1 – active)
k.out	Signals the beginning of the Output block processing
out.b	Represents the Output block state (0 – standby ; 1 – active)

Table 1 - Signal Description

	Time Interval (µs)	Description
t_1	13.085	Processing Time
t_2	10.4	Arithmetic Block Activations Time Interval
t_3	10.4	Output Block Activations Time Interval

Table 2 - Time Intervals

To measure the input/output delay, a 1KHz sinusoid was introduced at an input channel and forwarded to a certain output. Measuring the phase difference, a delay of 250μs was obtained. The input/output delay is even smaller than this value because the low-pass filter introduces a phase delay to the 1KHz sinusoid used to determine this value. This time interval corresponds to the processing time added to the conversion duration. The power consumption of the system was another measured parameter. It was detected a maximum of 600mA. This value was obtained with all outputs carrying a signal introduced in one of the input channels. Finally, a spectral analysis was done and the harmonic distortion and noise were measured. A 20KHz cut frequency was obtained. The total harmonic distortion plus noise (THD+N) is equal to 0,09% (Vin=1,28V @1KHz).

Evaluating the FPGA, it is verified that few resources are allocated to implement this project. Sixteen of the twenty embedded multipliers are used to generate two output samples at each cycle iteration on the arithmetic block's FSM. This value can be reduced from 16 to 8 by simply generating one instead of two samples per cycle. There were only used four of the twenty existing memory banks. As for Look Up Tables (LUT) and Flip Flops, the allocated resources are approximately 30% of the Spartan3E-500 FPGA according to the value presented by Xilinx ISE2.0 where the algorithm was synthesized.

5. Conclusions

MIAUDIO was successfully implemented (Fig. 8). A real-time multichannel diffusion system was created with a very compact and innovative architecture. A *low-cost* solution was achieved and its development time was relatively short.

Since the digital audio mixture is made in hardware, the computer that defines the parameters of the matrix has most of its resources free to engage in other possible tasks like producing effects over the audio signals, masterization, video synchronization, etc. This system is highly reconfigurable and new functionalities can easily be introduced without having to change the core of the system.

The obtained results were quite good given that the input/output delay is extremely low and that the signal's quality is assured. The *know-how* contained in this project also allows the development of other audio systems like a digital mixing surface.

References

[1] http://143.117.78.181/main.php?page=soniclab.
[2] http://www.audiosynth.com/scfaq.html.
[3] http://www.beast.bham.ac.uk/.
[4] http://www.beast.bham.ac.uk/about/meet.shtml.
[5] http://www.computermusic.org/members_only/array_issues/spring98/sw_reviews.html.
[6] http://www.digidesign.com/index.cfm?itemid=4892.
[7] http://www.digidesign.com/index.cfm?navid=349&langid=100&itemid=33116.
[8] http://www.rane.com/par-c.html#xlr.
[9] Burr-Brown. DAC8534,Quad Channel, Low Power, 16-Bit, Serial Input,Digital-to-Analog Converter, September 2002.
[10] Burr-Brown. PCM1802, Single-Ended Analog-Input 24-Bit, 96-KHz Stereo A/D Converter, January 2005.
[11] Digilent. Digilent Nexys2 Board Reference Manual, June 2008.
[12] Jonty Harrison. Diffusion: theories and practices, with particular reference to the beast system. http://cec.concordia.ca/econtact/Diffusion/Beast.htm.
[13] James McCartney. A new real time synthesis language. http://www.audiosynth.com/icmc96paper.htm
[14] Xilinx. Spartan-3E FPGA Family: Complete Data Sheet, April 2008.
[15] Digilent (September 2004). Digilent USB 2 Module Reference Manual.

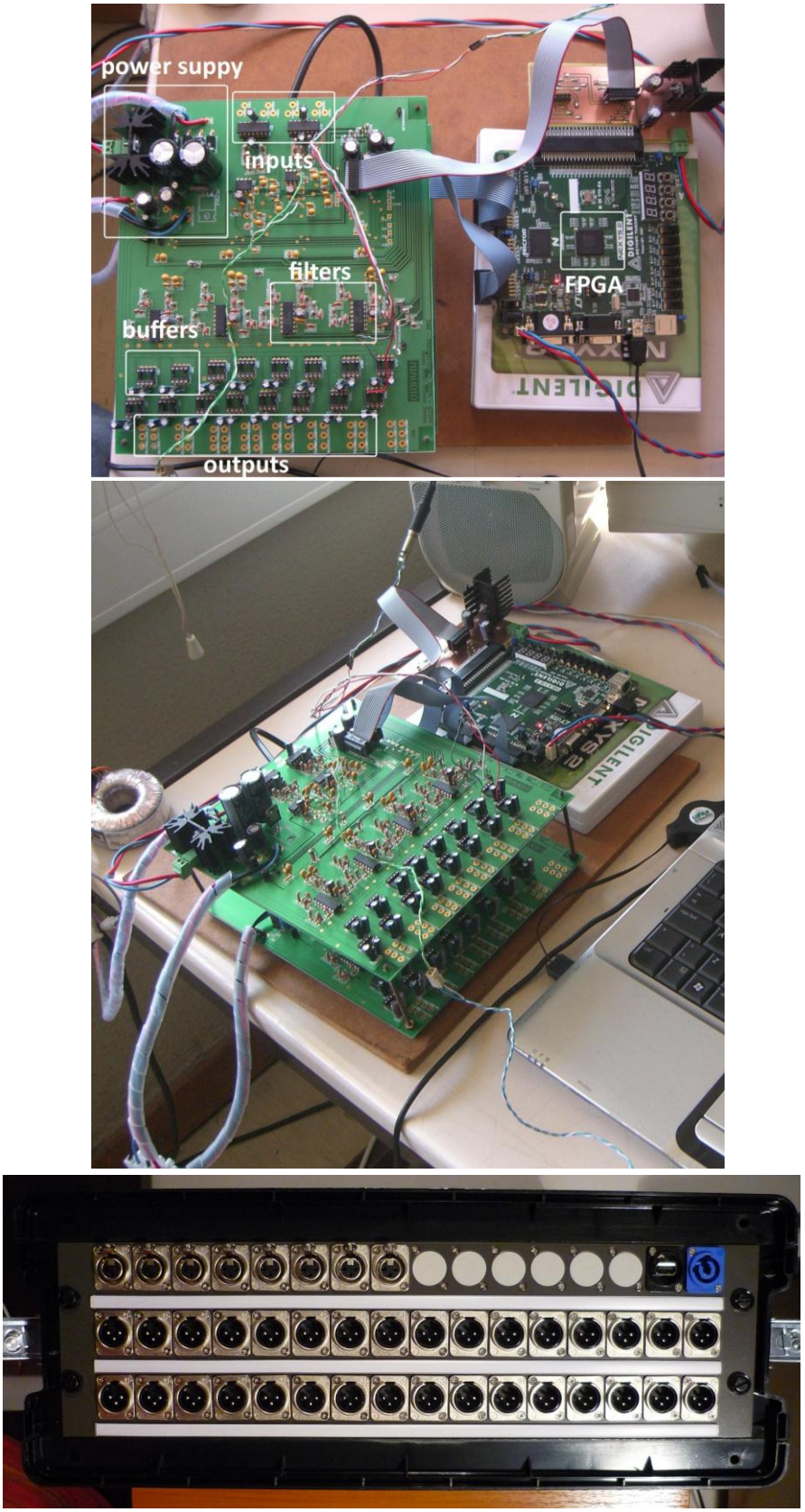


Fig. 8 – MIAUDIO

Real-time Optical-Flow estimation in FPGA*

João Pedro Santos, José Carlos Alves
{*jprosantos@gmail.com, jca@fe.up.pt*}
FEUP - Faculdade de Engenharia da Universidade do Porto
INESC Porto

Abstract

The extraction of movement information (or optical flow) from a video sequence demands for high computing power when needed in a real-time environment for accurate estimation of movement with sequences of high resolution images. Although present personal computers provide enough power to accomplish this task, embedded computing platforms based on low-power and low-performance CPUs cannot fulfill the real-time requirements of this computation, even for moderate resolution images. This is particularly interesting for various categories of robotic applications where computing power and consequently energy is constrained by physical space, weight or autonomy requirements. This paper presents the design and implementation of a custom designed optical flow estimator in a XILINX Spartan 3E FPGA. The system is aimed to interface to an embedded processor, providing the final optical flow estimation data to a software application. The implementation results have shown that the hardware system is able to process at real time VGA video sequences.

1. Introduction

The task of movement prediction is made possible by applying image processing techniques which have been largely implemented in software running in conventional processors. However, those implementations are not able to compute such data within the time-frame that current high-resolution/quality multimedia applications demand, when running on general purpose hardware architectures used by desktop PCs, for instance. Thus, the usage of dedicated hardware capable of computing fast motion prediction can improve significantly the performance of such software implementations.

Existing benchmark references on current hardware implementations of Optical Flow estimators indicate that there is room for some further improvements. The evolution of FPGA (Field Programmable Gate Array) platforms, with greater reconfigurable areas and higher speeds, offer now sophisticated platforms in which such custom architectures may be accommodated. Additionally, the ability to integrate a RISC microprocessor, capable of running Linux distributions, allows creating an interesting mix up of dedicated IP cores running under to control of an operating system (such as Linux), which is useful for robotic applications development, for instance.

*This work is funded by FCT (*Fundação para a Ciência e Tecnologia*), project PTDC/EEA-ELC/71556/2006

2. Visual Perception

The concept of Optical Flow (OF) concerns the subject of motion perception which is, itself, part of the study of visual perception. Optical Flow maps derive from Gibson's [1] concept of optic arrays added with speed significance. Throughout the years of experimentation many OF patterns have been studied and associated to different types of movement, fact that has simplified the task of OF interpretation. Nevertheless, it is important to stress that optical flow estimation does not intend to be enough to emulate the human visual perceptive system.

3. Optical Flow Estimation

There are four main general types of algorithms which are based on gradient, correlation, energy and phase calculation. In the course of the research, three specific techniques were studied, namely Horn & Schunck, Lucas & Kanade and Camus algorithms.

The use of gradients to calculate optical flow requires some constraints to be met. These were first identified by Horn & Schunck [2] in what is considered the reference publication of the subject. These conditions guarantee that image brightness is differentiable in every pixel.

Lucas & Kanade method applies the same constraints to a local set of pixels, in order to calculate OF, while Horn & Schunck provides a global OF solution.

Camus approach [3] results in a correlation based method, which according to [4], is a robust and one the fastest general purpose implementations of optical flow algorithms. It is essentially an upgraded block-matching technique that determines a measure of coincidence for a window of $(2\eta+1)\times(2\eta+1)$ pixels of likely displacement. The η parameter is related to the scene's maximum expected motion, as it defines the radius of the search windows used by the algorithm. The matching is accomplished by *moving* the patch of pixels which form the reference window in all possible displacements. By devising a simple matching technique as, for example, the sum of all differences (SAD), the most probable displacement can be determined.

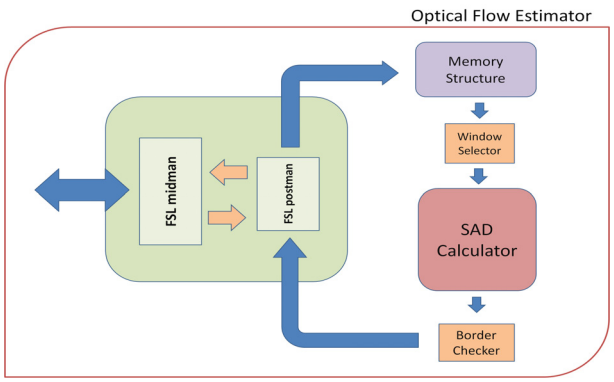


Figure 1. Optical Flow core elements.

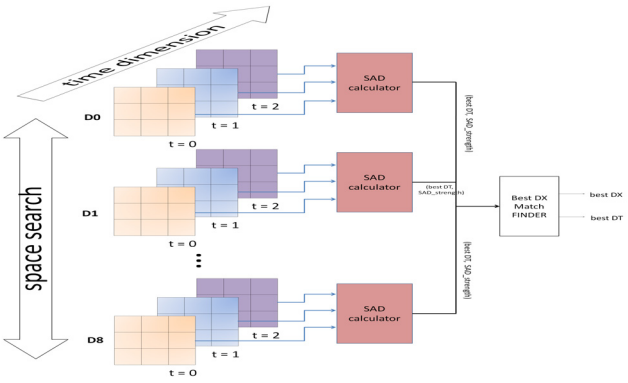


Figure 2. Organization of the SAD calculation engine.

A survey presented in [4] concludes that gradient based algorithms are generally more accurate than correlation based ones, at the cost of speed performance. Horn & Schunck is considered to be the best in terms of accuracy but the worst in speed. Camus approach is highly concerned with speed but presents an error in direction that is always above ten degrees [4] which, in some applications, may not be acceptable.

4. Estimator Implementation

The selected hardware platform was a SUZAKU-S SZ130-U00 which includes a Xilinx XC3S1200E FPGA and a default hardware project configuring uCLinux to run on MicroBlaze processor. This platform was chosen because it has several features interesting for embedded computing in robotics applications: small physical size, low power consumption and a significant part of free FPGA resources, after the implementation of the base MicroBlaze system. The objective of this work was to evaluate the feasibility of attaching a custom designed Optical Flow hardware estimator to a MicroBlaze processor included into this platform, to handle images received directly from low cost VGA digital cameras. As accuracy was considered to be less important than speed, in the scope of the target robotic applications, the Camus algorithm was selected.

This project was developed using the Xilinx development tools, ISE (Integrated Synthesis Environment) and XPS (Xilinx Platform Studio), version 10.1.03. The project was designed at the register-transfer level using the Verilog HDL.

A suitable setup for this algorithm uses a 3×3 pixel grid. The employed time *depth* by Camus in [3] suggests $S = 3$ frames. However, Barron *et al* [5], suggests that the ideal time search is close to 10 frames. The designed OF core is constituted by the SAD calculator, the FSL subsystem (that implements the interface to the MicroBlaze processor), the memory structure and some side modules, as shown in figure 1.

The *FSL midman* module is responsible for translating and processing commands received from the main processor *via* FSL. The function of the *FSL postman* module is to retrieve information from the surrounding modules and forward it to *FSL midman*. The *memory structure* implements three long shift-register structures that maintain five lines of each frame t_0, t_1, t_2. The five image lines are necessary to perform a 3×3 pixel window search around 1 pixel vicinity of the original position of each image block. The search is performed in parallel between a block of the current image and the corresponding search areas (5×5 pixel) of the two previous images maintained in memory. These shift-registers were implemented using FPGA LUTs instead of flop-flops, exploiting this important feature of XILINX FPGAs. Module *border checker* implements a watch mechanism that permanently evaluates the present pixel position in the frame. As the system is set to work in a continuous *blind* mode, it is necessary to prevent capturing results from pixels which are less than 2 pixels away from the image border. The last module, *window selector*, selects from the data in the image shift-registers, the set of pixels that represents the various search windows, to be routed to the SAD calculator. represents an abstraction of the process of automatically generating the candidate windows, by virtually displacing them in the sampling process. Finally, the *SAD* module fully exploits the data parallelism, computing in parallel the best match in all dimensions. The present reference window (current frame) is placed side by side with all the 9 possible displacements referring to the two previous frames (t_1 and t_2), using ±1 pixel displacement along both directions. Figure 2 illustrates the organization of the SAD module. The whole circuit is fully combinational and is composed of 9 SAD calculators, one for each of the 9 pixel displacements in the ±1 pixel search window. An additional circuits determines the best match of all the SAD values and selects the corresponding pixel displacement.

5. Implementation Results

This work has resulted in an implementation of the intended architecture for the optical flow estimator. These results were achieved by making plain usage of the FPGA resources having in mind the technological limitations of the implementation platform selected. The variety of tests performed using this architecture assure the expected accuracy and system behavior. In [6], it is available a C++ application which acts as the proof of concept for the correlation

978-972-789-304-1 REC'2010

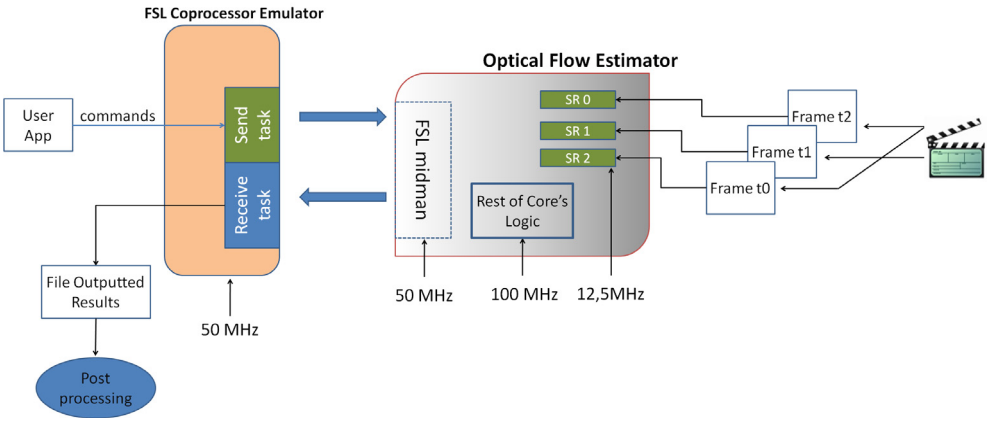


Figure 3. The simulation framework.

algorithm in which the estimator architecture was based. Thus, none of the conducted test sequences are aimed at quantifying the estimator's performance. McCane et al [7] were used for benchmarking reference.

The final hardware architecture is designed to be easily extrapolated to other systems or hardware platforms with higher slice, LUT or memory capacity, leaving it as object of future work. The estimator has the following general setup:

1. Search window of 3×3 pixels, displacement of ± 1 pixel;
2. Frame history S = 2;
3. Embedded memory structure of shift-register type of size $5 \times 640 \times (S+1)$;
4. Test frames are directly inputted to the SR-memory;
5. Resulting flow is visible after offline processing (by a virtual human interface).

Although these features do not allow attaining the initial performance markers, they are conceptually sufficient to demonstrate its functionality. Features 1 to 3 are parameter dependent, thus easily adaptable. The following require additional external memory and internal FPGA capacity.

The system validation was performed using a Verilog simulation model, as shown in figure 3, which implements the OF core and FSL communication system. An emulator of the MicroBlaze FSL bus translates user commands into the appropriate FSL commands, as issued through the MicroBlaze FSL ports. The image to use as stimuli is partitioned into the individual frames that are sent in sequence to the image ports of the simulation model, with 0 (the current frame), 1 and 2 frame delay. In a real hardware system, this could be easily implemented by intercalating appropriate FIFO memories between the camera and the computing FPGA, instead of sharing the system memory with the main processor. Further off-line applications use this information to create the movement maps with intensity and direction information.

The simultaneous implementation of the OF core and MicroBlaze does not fit within the FPGA's resources, even using the minimum algorithm settings (but keeping the VGA format specification). However, a standalone implementation of the whole system was completed for the target FPGA, without including the Microblaze processor, and considering an external memory system capable of delaying the two complete frames, as described above. This implementation uses 68% of LUTs and 36% of flip-flops, occupaying 83% of the FPGA slices. The whole system can run with a clock frequency of 100 MHz, meeting the 30 frame-per-second real-time constraint.

6. Experimental Results

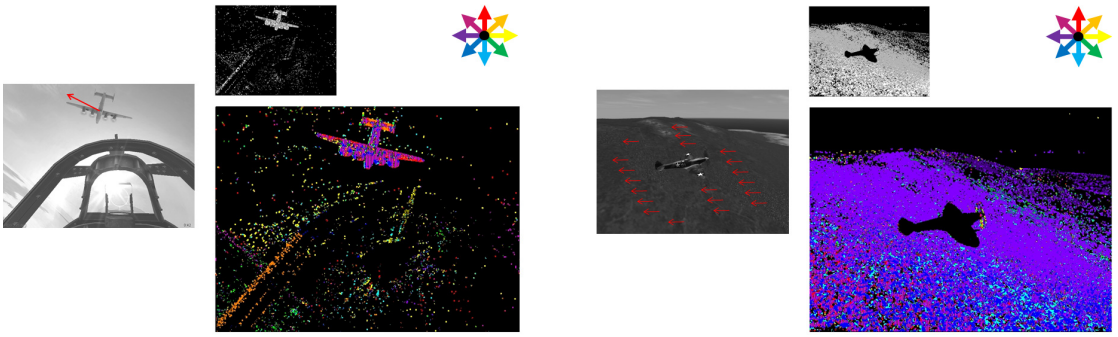
The following results were obtained using the simulation model. All the test cases aim to assure that the estimator characterizes correctly the movement patterns, within the full 640×480 frame resolution and multiple known optical flow patterns. The intensity (smaller frame) and directions (bigger frame) maps, are the result of off-line post-processing tasks. The scene's movement patterns are drawn on the left frame. These results can also be found at [8].

Test sequence 4(a) was designed to evaluate system performance, in relatively good detection conditions, for object tracking purposes. It shows the calculated flow to be accurate enough to a clear a clear map in which the object stands out.

Test sequence 4(b) puts a fixed camera on the main object and watches the background move in a continuous direction. The object is perceived as static and removed from the output. The visible pattern stands for the background movement of the landscape, as perceived by the observer. In addition to this, the sky remains static as expected.

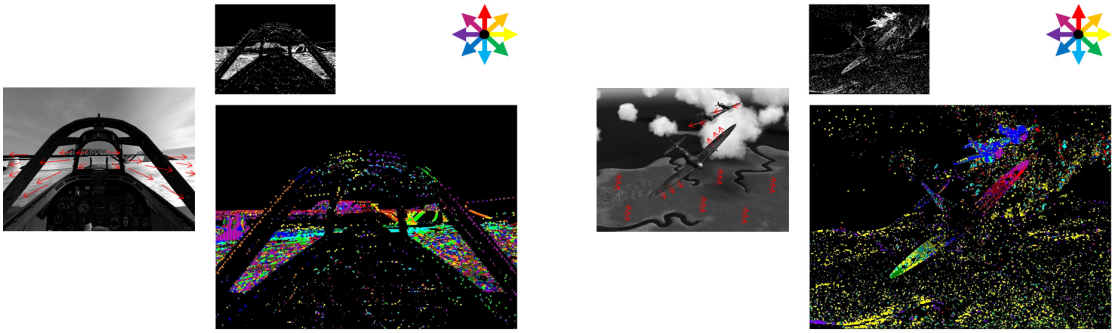
Test sequence 4(c) illustrates a typical expansion pattern where a structure is rapidly closing in on the observer. There are two distinguishable plains: the ground whose movement characterization is fuzzy and intense and the sky which is pictured static.

Test sequence 4(d) includes multiple objects moving with various velocities and with different directions. There are also multiple contrast variations for each object together with slow background movement as the camera follows the turn of the object in the center. The outputted data suffers from some noise, but showing that the estimator determines accurately most of the scene flow.



(a) Object tracking by movement analysis.

(b) Detection of background movement.



(c) Expansion patterns.

(d) Complex environment with multiple patterns.

Figure 4. Results obtained with the OF estimator simulation model.

7. Conclusion

This system is capable of processing 640×480 frames in real-time (30 frames/s), using a search window of 3×3 pixels and $S = 2$ frames, with an expectable maximum error of about 10^{o} in direction. This system is designed to operate in conjunction with an embedded MicroBlaze CPU communicating through the FSL bus. The Optical Flow core is capable of running at a clock frequency up to 100 MHz, interfacing the microprocessor at 50 MHz. The Optical Flow core can be accessed from as application running in the MicroBlaze by calling a reduced set of interface functions. This system was implemented on a XILINX Spartan 3E 1200 FPGA included in a SUZAKU-S SZ130-U00 hardware platform.

There is space for improvement in terms of algorithm performance, although it essentially depends on the available logic resources in the FPGA. The estimator can be expanded to pursue better results in real test cases, by employing wider than 3×3 pixels reference areas and by extending the time search.

The developed estimator may be employed to retrieve OF data allowing to compute more complex perceptions out of video frames such as obstacle recognition, time-to-contact, and navigation in general. Any of these applications represents a possible post-processing application which could be designed based on the existing framework.

References

[1] J. J. Gibson, *The Ecological Approach to Visual Perception*. Lawrence Erlbaum Associates, 1979.

[2] B. K. P. Horn and B. G. Schunck, "Determining Optical Flow," 1981.

[3] T. Camus, "Real-time quantized optical flow," in *Proc. Computer Architectures for Machine Perception CAMP '95*, 18–20 Sept. 1995, pp. 126–131.

[4] A. H. Liu, H. Liu, T. hong Hong, and M. Herman, "Accuracy vs. Efficiency Trade-offs in Optical Flow," in *Computer Vision and Image Understanding*. Academic Press, 1996, pp. 271–286.

[5] J. L. Barron, D. J. Fleet, and S. S. Beauchemin, "Performance of optical flow techniques," *International Journal of Computer Vision*, vol. 12, pp. 43–77, 1994.

[6] B. McCane. (2009, June) Graphics and Vision Research Laboratory. Department of Computer Science, University of Otago. [Online]. Available: http://www.cs.otago.ac.nz/gpxpriv/vision.html

[7] B. McCane, K. Novins, D. Crannitch, and B. Galvin, "On benchmarking optical flow," *Comput. Vis. Image Underst.*, vol. 84, no. 1, pp. 126–143, 2001.

[8] J. P. Santos. (2009, July) Implementation OF Algorithms in FPGA Platforms with Embedded CPU. [Online]. Available: http://sites.google.com/site/ofinfpgawithembeddedcpu/

Interlayer Intra Prediction Architecture for Scalable Extension of H.264/AVC Standard

Thaísa Silva, Luís Cruz
Telecommunications Institute – University of Coimbra, Portugal
lealth@gmail.com, lcruz@deec.uc.pt

Luciano Agostini
Group of Architectures and Integrated Circuits – Federal University of Pelotas, Brazil
agostini@ufpel.edu.br

Abstract

This paper presents an architecture for the interlayer intra prediction mechanism of the scalable extension of the H.264/AVC video coding standard. This hardware module is used between spatial layers in the scalability process. The architecture of the interlayer intra prediction is composed of two main modules: a deblocking filter and an upsampling module. These modules were described in VHDL and synthesized targeting Stratix III and Stratix IV Altera FPGAs device, respectively, and they were validated with the ModelSim tool. The results obtained through the synthesis of the deblocking filter architecture show that the sample filtering order and the use of four concurrent filter cores reduce by almost 25% the number of cycles used in the filtering process when compared to related works. Moreover, this architecture is able to filter up to 130 HDTV frames per second. The synthesis results presented for the complete upsampling architecture show that this architecture is able to achieve processing rates of 384 VGA frames per second. With these results the interlayer intra prediction architecture proposed reached high enough processing rates to allow processing VGA or HDTV video in real time.

1. Introduction

The increasing number of different types of devices used by millions of users that are able to handle digital video, ranging from cell phones to high-definition televisions, creates a problem if we desire to code and transmit a single video stream to be used by all these types of devices. The scalable extension to the H.264/AVC (Advanced Video Coding) standard [1] was developed to cover this wide range of video applications defining the syntax and semantics of a scalable video bitstream. From this bitstream, different decoded video signals with distinct spatial resolutions, frame rates, and/or bit rates can be extracted.

The H.264 Scalable Extension was added to the H.264/AVC standard as Amendment 3 [2, 3]. This extended standard (also called as H.264/SVC – Scalable Video Coding) supports scalabilities in the temporal, spatial and quality dimensions and structurally it is composed of one base layer coder, which is compliant with the H.264/AVC [3] coder and one or more enhancement layers. The enhancement layers' data are coded based on predictions formed by the base layer frames and by the previously encoded enhancement layer frames.

In the spatial scalability mode each enhancement layer can use interlayer prediction mechanisms which were added to the standard to minimize the redundant information present in different layers [3]. These mechanisms are: interlayer intra prediction, interlayer motion prediction and interlayer residual prediction from its lower layer [3]. In addition, the enhancements layers can also be encoded in an AVC compliant mode, which is independent of the interlayer prediction coding tools.

This work is focused on the implementation of the upsampling and deblocking operations of the spatial interlayer intra predictor. The architecture proposed considers the dyadic case (when the resolution doubles horizontally and vertically between layers) and the base layer resolution was defined as QVGA (320x240 pixels) and the enhancement layer was defined as VGA (640x480 pixels). These resolutions were adopted due to the performance limitations of other modules of the decoder H.264/SVC.

This paper is organized as follows. In section 2 the Interlayer Intra Prediction is presented. Section 3 outlines the proposed architectures. In section 4 the

synthesis results are presented and in section 5 the related works are addressed. Finally, section 6 presents the conclusions of this work.

2. Interlayer Intra Prediction in the H.264/AVC Scalable Extension

From the interlayer prediction, the base layer information is adaptively used to predict the information of the enhancement layer. This increases the coding efficiency of the enhancement layer.

Fig. 1 shows a typical coder structure with two spatial layers where the base layer bitstream is generated to be compatible with the non-scalable H.264/AVC standard. The interlayer intra predictor located between the base (*Layer 0*) and enhancement (*Layer 1*) layers generates a prediction of the higher layer information by upsampling the image data from the corresponding region in the lower layer. However, to reduce block-edge discontinuities in the image data caused by high quantization steps during the coding at the reference layer, the upsampling operation is preceded by a filtering operation performed by the deblocking filter. This filter is very similar to the deblocking filter used at the end of the coding/decoding process of the H.264/AVC without scalability, even though it performs a different calculation for the boundary strength.

The upsampling is responsible to adapt the coding information in the lower resolution layer to the higher layer resolution. It is applied when the prediction mode of a block is interlayer and the corresponding block in the reference layer has been encoded using intra prediction.

3. Designed Architecture

The interlayer intra prediction architecture is composed by the deblocking filter and upsampling architectures, whose details follow:

a) **Deblocking Filter Architecture**: The filter is applied across the horizontal and vertical boundaries of each 4x4 block of the luminance and chrominance macroblocks, as shown in Fig.2.

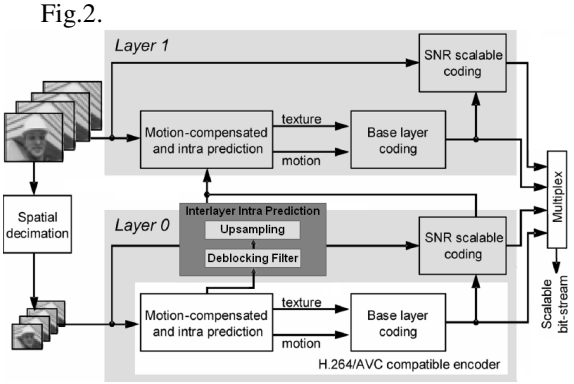


Fig. 1. Typical coder structure for the scalable extension of H.264/AVC.

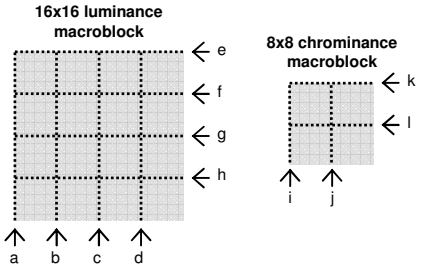


Fig. 2. Filtering ordering of the luminance and chrominance borders.

The filtering operation is performed according to the following steps: filtering vertical edges of luminance macroblock (MB) (*a*, *b*, *c* and *d*, in Fig.2); filtering horizontal edges of luminance MB (*e*, *f*, *g* and *h*, in Fig.2); filtering vertical and filtering horizontal edges of chrominance MBs (*i* and *j* and *k* and *l*, in Fig.2). Each filtering operation modifies up to three pixels on each side of the edge and involves four pixels of each of two neighboring blocks that are filtered. All the filtering orders found in the literature were performed at block level, i.e., the calculations of a border between two 4x4 blocks are performed serially by the same filter and each filtering between two blocks starts just when the filtering of all the LOPs (Line of Pixels) of the predecessor block (left) finishes. This work proposes an efficient processing order at sample level, instead of block level. The complete filter architecture is presented in Fig. 3. This architecture is composed of one *bS* calculator module, one thresholds calculator module, one *c1* calculator module, eight transpose matrices (*T1-T8* in Fig. 3) and four filtering cores (*F1-F4* in Fig. 3). The *bS* calculator defines the filtering strength based on some coding information and the threshold calculator defines the values of α and β based on the quantization parameters of the two blocks which are being filtered. The *c1* calculator calculates a clipping value that is used in the filtering process based on the filtering strength and on the threshold values. Each transpose matrix stores the samples of a full block, in addition to its coding information. The filtering cores perform the filtering operations using the samples and the values of *bS*, thresholds (α and β) and *c1*, which were previously calculated. The architecture operates in a pipelined structure that performs four concurrent filtering operations.

b) **Upsampling Architecture**: In the upsampling module a multiphase 4-tap filter is applied to the luma components and a bilinear filter is applied to the chroma elements. The filters are applied first horizontally and after vertically. The use of different filters for luma and chroma is motivated by complexity issues.

978-972-789-304-1 REC'2010

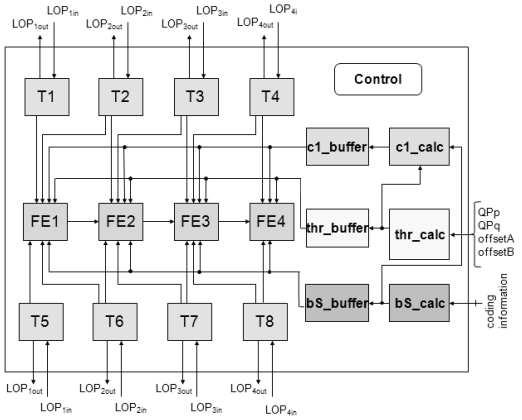


Fig. 3. Complete Architecture of the Interlayer Deblocking Filter.

In the H.264 Scalable Extension, the upsampling uses a set of 16 filters, where the filter to be applied is selected according to the upsampling scale factor and the sample position.

The luminance filters are defined by equations (1) and (2) and equations (3) and (4) are applied to the chrominances filters.

$$S4 = (-3.A + 28.B + 8.C - D) >> 5 \quad (1)$$

$$S12 = (-A + 8.B + 28.C - 3.D) >> 5 \quad (2)$$

$$S4 = (24.A + 8.B) >> 5 \quad (3)$$

$$S12 = (8.A + 24.B) >> 5 \quad (4)$$

Fig. 4 shows the architecture for the complete upsampling module, wich is composed two luminance filters (horizontal and vertical – Luma H Filter and Luma V Filter, respectively) and the two chrominance filters (horizontal and vertical – Chroma H Filter and Chroma V Filter, respectively). Also are represented memories MEM IN, which work as input buffers for the luminance and chrominance filters and memories MEM H1 and MEM H2, used as ping-pong transpose buffers between the horizontal and vertical filters of luminance and chrominance and the clipping operators (Clip, in Fig. 4). To simplify the figure the memory address registers their multiplexers and respective control signals were omitted. The filters for the luminance and chrominance were designed through algebraic manipulations of equations showed in (1), (2), (3) and (4) to replace multiplies by shifts, resulting in equations (5), (6), (7) and (8). These manipulations were performed in order to save hardware resources.

$$S4 = (-2.A - A + 16.B + 8.B + 4.B + 8.C - D) >> 5 \quad (5)$$

$$S12 = (-A + 8.B + 16.C + 8.C + 4.C - 2.D - D) >> 5 \quad (6)$$

$$S4 = (16.A + 8.A + 8.B) >> 5 \quad (7)$$

$$S12 = (8.A + 16.B + 8.B) >> 5 \quad (8)$$

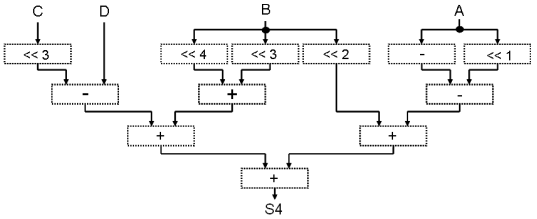


Fig. 5. Internal architecture of the luminance filter of index 4.

Fig. 5 shows the architecture defined to implement equation (5), which represents the luminance filter with index 4. The others architectures were designed in a way similar to presented in Fig. 3.

4. Results and Comparisons

The interlayer intra prediction architecture proposed in this paper takes 53 cycles to filter a complete macroblock, which is about 25% less than the best result of the previous solutions with the same number of filtering cores [4]. These good results are obtained through the use of an efficient filtering order combined with a new architectural solution. Tab. 1 shows other solutions listing the number of cycles necessary to filter one macroblock and the size of temporary memory used.

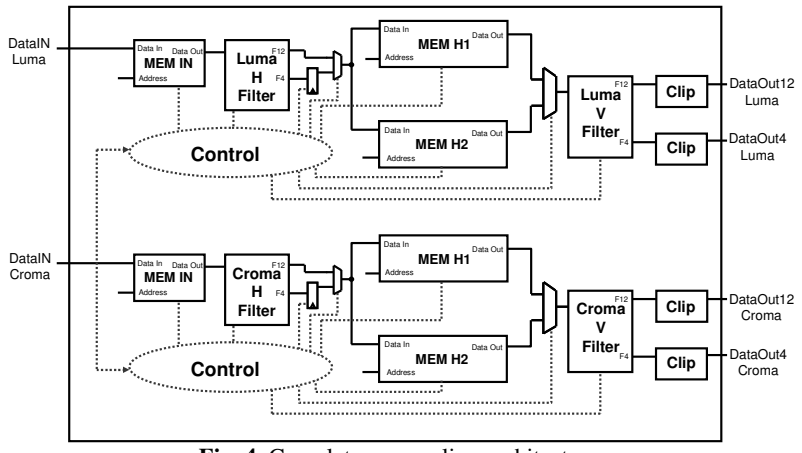


Fig. 4. Complete upsampling architecture.

The filter architecture proposed was described using VHDL and synthesized for Altera Stratix III FPGA (EP3SL50F484C2 device). The core filter uses 737 ALUTs and the complete architecture uses 7,868 ALUTs. The architecture is able to run at around 270 MHz and considering a HDTV (1920 x 1080 pixels) resolution, the processing rate of the designed filter is around 130 frames per second. This frame rate outperforms the minimum real time HDTV filtering requirements, allowing the use of a lower filter operating clock frequency when processing VGA video.

Table 1. Comparison between processing orders

Filtering Order	Cycles per MB	Filter Cores	Memory Needed (bytes)
H.264/AVC	192	1	512
Khurana [6]	192	1	128
Sheng [7]	192	1	80
Li [8]	140	2	112
Ernst [4]	70	4	224
Our order	**53**	**4**	**128**

The upsampling architecture was initially synthesized to Altera Stratix III FPGAs, but as it was not possible to synthesize the complete architecture of the upsampling, it was necessary to synthesize the architecture targeting devices of the Altera Stratix IV FPGAs. Tab. 2 summarizes the upsampling synthesis results. Two timing models were used to evaluate the designs. The first frequency result presented in the Tab. 2 was obtained using the pessimistic model at 85ºC, also called "Slow 900mV 85C Model" (Model 1) and the second frequency result was obtained using the pessimistic model at 0ºC, also called "Slow 900mV 0C Model" (Model 2). From the results presented in Tab. 2 it was possible to calculate the processing rates, which for VGA resolution is at least (worst case) 384 frames per second. This is the first design reported in the literature for the upsampling filter of the H.264/SVC standard then it was not possible to compare this work with others designs.

5. Conclusions

This work presented the design of an architecture for the deblocking filter module and other for the upsampling module, which compose the interlayer intra prediction architecture used in the scalable video coding according to the scalable extension of the H.264/AVC standard. These modules were designed in the context of a codec supporting two spatial dyadic layers with resolutions QVGA (base layer) and VGA (enhancement layer).

With relation to the results presented by the deblocking filter architecture it is possible notice its advantages over similar published designs, outperforming the best one by 25%. Moreover, this architecture reached a maximum operation frequency around 270 MHz, which means that it is able to process 130 HDTV frames per second. The upsampling complete architecture presented an operation frequency of 119.5 MHz in the worst case, allowing a processing rate of 384 VGA frames per second.

Both architectures presented quite satisfying results that outperform the minimum requirements to decode high definition videos in real time.

References

[1] INTERNATIONAL TELECOMMUNICATION UNION. ITU-T Recommendation H.264 (11/07): advanced video coding for generic audiovisual services. [S.l.], 2007.

[2] T. Wiegand, G. Sullivan, J. Reichel, H. Schwarz and M. Wien, ISO/IEC JTC 1/SC 29/WG 11 and ITU-T SG16 Q.6: JVT-W201 'Joint Draft 10 of SVC Amendment,' 23th Meeting, San Jose, California, April 2007.

[3] H. Schwarz, D. Marpe, and T. Wiegand, "Overview of the Scalable Video Coding Extension of the H.264/AVC Standard," IEEE Transaction on Circuits and Systems on Video Technology, vol.17, no.9, Sep. 2007.

[4] E. Ernst, "Architecture Design of a Scalable Adaptive Deblocking Filter for H.264/AVC", MSc Dissertation, Rochester, New York, 2007.

[5] T. Wiegand, G. Sullivan, and A. Luthra, "Draft ITU-T Recommendation and final draft international standard of joint video specification (ITU-T Rec.H.264|ISO/IEC 14496-10 AVC)", 2003.

[6] G. Khurana, T. Kassim, T. Chua, and M. Mi, "A pipelined Hardware Implementation of In-loop Deblocking Filter in H.264/AVC", IEEE Transactions on Consumer Electronics, 2006.

[7] B. Sheng, W. Gao, and D. Wu, "An Implemented Architecture of Deblocking Filter for H.264/AVC", International Conference on Image Processing, 2004.

[8] L. Li, S. Goto, T. Ikenaga, "A highly parallel architecture for deblocking filter in H.264/AVC", IEICE Transactions on Information and Systems, 2005.

Table 2. Synthesis results for the proposed architecture for the upsampling module.

	Luma Core	Chroma Core	Luma	Chroma	Complete Upsampling
Model 1 Frequency (MHz)	151.42	381.68	137.61	190.99	119.5
Model 2 Frequency (MHz)	161.39	406.01	146.76	202.59	127.42
ALUTs	154	42	1,267	759	2,024
Dedicated Logic Registers	66	40	577	454	1,032
Memory Bits	-	-	5,222,400	1,288,800	6,451,200

Selected Device: Stratix IV EP45SGX530HH35C3

Sessão Posters

Introdução: Arnaldo Oliveira
Universidade de Aveiro / IEETA

Utilização de Lógica Programável no Ensino de Sistemas Digitais no IPS/ESTSetúbal

Ana Antunes, José Sousa
Instituto Politécnico de Setúbal/ ESTSetúbal
aantunes@est.ips.pt, jsousa@est.ips.pt

Abstract

Este artigo descreve e contextualiza a utilização de lógica programável no ensino de Sistemas Digitais na Escola Superior de Tecnologia de Setúbal do Instituto Politécnico de Setúbal (ESTSetúbal/IPS).

1. Enquadramanto

A ESTSetúbal é uma das cinco escolas do Instituto Politécnico de Setúbal e iniciou a sua actividade lectiva no ano de 1988/1989. A oferta da ESTSetúbal/IPS recai sobre 7 cursos de licenciatura e 8 de mestrado para além de cursos de formação e pós-graduação. A ESTSetúbal/IPS desenvolve ainda investigação e, cada vez mais, aposta na formação pós-graduada e contínua.

As temáticas associadas aos sistemas digitais vão desde a lógica aos sistemas embebidos passando pelos microprocessadores e arquitectura de computadores. Ao nível mais básico são leccionadas duas unidades curriculares designadas Sistemas Digitais (SD) e Sistemas Digitais II (SDII). Estas unidades curriculares integram os planos de estudos dos cursos de Eng. Electrotécnica e de Computadores (ramos de Electrónica e Computadores e de Electrónica e Telecomunicações) e Eng. Biomédica (ramo de Bioelectrónica), respectivamente no 1º e 2º anos. É na vertente da prática laboratorial da unidade curricular SDII que o recurso à lógica programável tem mais expressão.

O conteúdo programático de SDII inclui tópicos como: circuitos sequenciais assíncronos, memórias, circuitos sequenciais microprogramados e lógica programável [1]. Ao longo dos últimos três anos a média de alunos inscritos nesta unidade curricular é de 55, divididos em turmas de laboratório com 16 alunos que funcionam em grupos de 2 alunos.

Na ESTSetúbal/IPS o ensino das engenharias tem uma forte componente prática/laboratorial e a distribuição das horas pelas várias componentes da unidade curricular a leccionar reflecte essa filosofia. Assim SDII tem uma carga semanal de 6 horas de aulas presenciais divididas em 2 horas de aulas teórico-práticas e 2 aulas de 2 horas de laboratório. Estão-lhe atribuídos 6 ECTS (*European Credit Transfer and Accumulation System*).

Nos laboratórios são realizados trabalhos de desenvolvimento temáticos em que se apresenta um problema cuja solução os alunos, de forma autónoma, devem conceber, desenvolver, apresentar e defender. O último desses trabalhos (projecto), de maior dimensão, visa a resolução de um problema realístico. É condição necessária para o sucesso na avaliação, que os circuitos desenhados funcionem satisfazendo as premissas impostas no enunciado sendo, também, estimulado o desenvolvimento de funcionalidades adicionais.

2. Utilização de Lógica Programável nos Laboratórios de SDII

Na ESTSetúbal/IPS optou-se pela utilização obrigatória de lógica programável apenas nos laboratórios da unidade curricular de SDII. Nos laboratórios da unidade curricular precedente, SD, desenvolvem-se pequenos trabalhos independentes, e um projecto autónomo final que são implementados apenas com recurso a componentes digitais básicos uma vez que, no plano de estudos dos cursos envolvidos, esta unidade decorre em paralelo com a primeira unidade curricular de Electrónica.

Em SDII são realizados trabalhos temáticos sobre máquinas de estados síncronas por realização directa (6 aulas) e máquinas de estados assíncronas em modo fundamental (7 aulas), sendo que o projecto consiste na realização de uma máquina de estados microprogramada que controlará os circuitos desenvolvidos nos trabalhos temáticos anteriores (9 aulas).

A realização de trabalhos com esta complexidade, especialmente o controlo de arquitecturas, é muito difícil e/ou penoso utilizando lógica discreta. De facto, essa abordagem limitaria a dimensão do projecto quer devido ao tempo envolvido na montagem, verificação e correcção de

problemas associados às ligações dos componentes, quer devido a questões de ordem mais prática como a portabilidade de um tal circuito. A ênfase dada à realização de circuitos mais complexos e realistas promove o desenvolvimento de competências, não só no que concerne aos tópicos leccionados nas aulas teórico-práticas mas, também, em aspectos complementares como as metodologias de projecto.

Actualmente o desenvolvimento dos circuitos é efectuado sobre a plataforma de *software ISE WebPACK 10.1* [2] da Xilinx. A escolha desta plataforma assenta essencialmente em dois aspectos: (1) a forte presença dos produtos Xilinx no mercado para o qual se estão a formar os alunos e (2) o facto da Xilinx disponibilizar aos alunos o acesso gratuito ao *software.*

Tirando partido das potencialidades de plataformas deste tipo, os alunos aprendem também a utilizar a simulação como uma ferramenta apropriada para o desenvolvimento de circuitos digitais. No entanto, como a ênfase é sempre posta na realização concreta de um circuito, utiliza-se também uma placa de desenvolvimento para implementar e testar os circuitos desenhados.

A placa de desenvolvimento utilizada em SDII é a *Basys* da Digilent [3] desenhada em torno de uma FPGA Spartan 3E da Xilinx. A escolha desta placa de desenvolvimento teve por base aspectos como: (1) baixo custo relativamente a outras soluções disponíveis no mercado, (2) elevado número de interfaces que permitem a integração com diversos outros circuitos e/ou dispositivos, (3) integração de interfaces com o utilizador que permitem a realização de trabalhos autónomos com recurso exclusivo à placa e (4) pequena dimensão.

Os trabalhos/ projectos desenvolvidos passam por todas as fases de desenvolvimento, começando na concepção do circuito, continuando no desenho do esquema e sua simulação sobre o ambiente de desenvolvimento da Xilinx, terminando com a síntese e teste do circuito já em *hardware*.

Complementarmente a utilização da ferramenta de desenvolvimento da Xilinx permite aos alunos o contacto com uma linguagem de descrição de *hardware*, neste caso o Verilog que não é formalmente abordada nas aulas teóricas. Os alunos aprendem a linguagem de modo autónomo através da exploração das funcionalidades da ferramenta de desenvolvimento e, no decurso do seu projecto, escolhem livremente quais os blocos a implementar a partir de esquemas ou, em alternativa, recorrendo ao Verilog.

3. Conclusão

A experiência desenvolvida na ESTSetúbal/IPS nos laboratórios de SDII ao longo dos últimos dois anos permite salientar alguns aspectos da utilização das plataformas de lógica programável.

O aspecto mais positivo prende-se com a possibilidade de apresentar aos alunos problemas mais complexos e realistas, sem o ónus das distracções derivadas das dificuldades inerentes à montagem de circuitos sobre plataformas de prototipagem como as *bread-boards*. É possível, por isso, promover o desenvolvimento de mais competências no processo de formação dos alunos por inclusão de mais conceitos e técnicas num mesmo projecto.

Outra mais valia da utilização de lógica programável nos laboratórios é a possibilidade, oferecida ao aluno, de desenvolver competências complementares, neste caso associadas à aprendizagem de Verilog.

Relativamente à plataforma de *software* adoptada, uma vez que a ferramenta de desenvolvimento da Xilinx está disponível gratuitamente: (1) promove a independência dos alunos relativamente ao espaço físico dos laboratórios de sistemas digitais e (2) estimula o desenvolvimento de trabalho autónomo.

O custo de montagem de um laboratório de sistemas digitais baseado em lógica programável pode ser visto como o principal impedimento a este tipo de abordagem. No entanto, na ESTSetúbal/IPS esse custo é encarado como um investimento viável uma vez que: (1) as placas de desenvolvimento podem ser utilizadas no âmbito de outros projectos e/ou unidades curriculares, o que resulta na diluição do custo, (2) a imagem dos cursos e, como tal, da ESTSetúbal/IPS é valorizada pela utilização e ensino de competências com tecnologias "de ponta" e (3) o investimento necessário para manter o *stock* de componentes avulso pode ser reduzido.

O impacto da adopção da lógica programável na taxa de sucesso escolar não foi um factor ponderado. Na opinião dos autores, o sucesso escolar não depende dos trabalhos de laboratório serem desenhados com vista à implementação sobre lógica programável ou discreta. A utilização da lógica programável vem, isso sim, contribuir para o alargamento das competências adquiridas pelos alunos melhorando assim a qualidade da componente laboratorial da unidade curricular. Este é um aspecto que poderá ter uma influência positiva no sucesso das unidades curriculares de sistemas digitais subsequentes.

References

[1] Página de Sistemas Digitais II. www.si.ips.pt/ests_si/

[2] Página Web do ISE WebPACK http://www.xilinx.com/support/download/index.htm

[3] Digilent Basys Board Reference Manual, Digilent, 2007.

LÓGICA PROGRAMÁVEL UMA NOVA ABORDAGEM NO ENSINO DA ELETRÔNICA DIGITAL NA DIREÇÃO DAS NOVAS TECNOLOGIAS DE AUTOMAÇÃO INDUSTRIAL

Cesar da Costa
Faculdade de Engenharia - Departamento de Mecânica
UNESP-Universidade Estadual Paulista Julio de Mesquita Filho
cost036@attglobal.net.

Resumo

A proposta deste trabalho, com base em um avanço tecnológico crescente, é apresentar e discutir o emprego de novas ferramentas computacionais no desenvolvimento e ensino da Eletrônica Digital. A tecnologia digital vem crescendo de forma exponencial, novos equipamentos e sistemas embutidos são especialmente projetados para trabalhar em aplicações de Automação Industrial. O projeto e a manutenção desses sistemas requerem uma grande demanda de mão de obra especializada. Dentro desse contexto, uma nova abordagem faz-se necessária, no sentido de modernizar o ensino de Eletrônica Digital nos cursos de Licenciatura de Engenharia das escolas tradicionais e das Escolas Politécnicas.

1. Introdução

Sempre que ocorre uma grande mudança na tecnologia, há um período durante o qual as instituições de ensino têm de decidir como e quando mudar a maneira como ensinam os assuntos relacionados às mudanças tecnológicas. Alguns se lembram da mudança das válvulas eletrônicas para os transistores e a maioria lembra-se da substituição dos circuitos transistorizados pelos amplificadores operacionais [1].

Recentemente, a tecnologia de sistemas digitais se moveu na direção da lógica reconfigurável. Muito pouco das novas tecnologias de automação industrial como projeto de robôs, sistemas de visão, inversores de frequência, PLCs (autômatos), sensores inteligentes, dispositivos FieldBus e DeviceNet, usam circuitos digitais convencionais ou microprocessadores tradicionais na implementação de suas funções internas de controlo. A maioria dos circuitos internos desses dispositivos está contida em um único dispositivo reconfigurável, FPGA (Field Programmable Gate Array) ou CPLD (Complex Programmable Logic Devices) [2].

2. Novas Tecnologias

Para aprender como criar esses "sistemas em chip", os futuros engenheiros têm de entender o funcionamento desses novos dispositivos, baseado em lógica reconfigurável, linguagens de programação de hardware VHDL ou Verilog e ferramentas de projeto como EDA (Eletronic Design Automation), por exemplo. Entretanto, atualmente na maioria dos cursos de Licenciatura de Engenharia das escolas tradicionais e nas escolas Politécnicas, essas novas tecnologias ainda não são devidamente ensinadas, perde-se um tempo precioso, em aulas teóricas e práticas (laboratório), na montagem de circuitos baseados em microprocessadores PIC, por exemplo, e na elaboração manual de programas em linguagem C.

A execução de um algoritmo de controlo num microprocessador tradicional depende de um software armazenado em memória, que será executado numa arquitetura tipo Von Neumann, por exemplo, com ciclos de busca e execução das instruções. Numa arquitetura baseada em lógica reconfigurável com FPGA, um algoritmo é implementado por hardware, sem a necessidade de ciclos de busca e execução de instruções. O problema básico a ser resolvido é a implementação de uma arquitetura eficiente, para execução desse algoritmo ao invés de compilá-lo para sua execução em uma CPU [3 e 4].

Uma das grandes vantagens da utilização de FPGAs nessas novas arquiteturas é a possibilidade

de se definir vários blocos de hardware, que operam em paralelo, aumentando muito a capacidade computacional de um autômato num controle industrial, por exemplo [5]. Já o ambiente de desenvolvimento além de ter o tempo e o custo reduzido em relação aos ambientes tradicionais de projetos, permite simular e testar rapidamente em campo o protótipo ou a versão final do hardware [6]. A figura 1 apresenta duas placas com eletrônica embutidada, baseada em lógica reconfigurável.

Figura 1 – Placas de controlo industrial com eletrônica embutida baseada em lógica reconfigurável.

3. Nova Proposta de Ensino de Eletrônica Digital

A proposta de uma nova abordagem no ensino da Eletrônica Digital tem como meta possibilitar que o estudante, futuro engenheiro, tome conhecimento das mudanças tecnológicas que estão ocorrendo no mundo, principalmente nos departamentos de projeto de grandes empresas como Siemens, GE Fanuc, Samsung, ABB, Yokogawa, Omron, Schineider, Smar, National, Analog Devices, Pentek, diversos fabricantes da indústria militar, indústria aeronáutica, indústria naval e a indústria automobilística, ou seja, a consolidação de novas tecnologias de projetar e implementar equipamentos com electrônica embutida, baseada em lógica reconfigurável.

Os sistemas de lógica reconfigurável são importantes, não só pelos ganhos de desempenho, mas também pela possibilidade de serem rapidamente atualizados e/ou reparados sem necessidade de substituição física, basta a sua reconfiguração por software [7].

4. Sugestões de Novos Currículos

A sugestão de novos currículos para o ensino da Eletrônica Digital tem por objetivo, contribuir para a modernização do ensino das disciplinas que compõe o currículo regular dos cursos de Licenciatura em Engenharia das Escolas tradicionais e as Escolas Politécnicas. E ainda, a formação de mão-de-obra especializada na área, de alta qualificação e capaz de um trabalho independente voltado quer para o ambiente universitário, quer para o ambiente industrial. Pretende-se assim que o futuro profissional possa continuar a desenvolver, aprofundar e aplicar os seus conhecimentos derivados do seu trabalho de projeto e aplicações, tornando-se um profissional de mais importância para as empresas onde decorre a sua atividade. Deve salientar-se que no atual panorama empresarial nacional (Portugal e Brasil), o número de pessoas com conhecimentos em lógica reconfigurável e ferramentas computacionais de projeto, sobretudo ao nível de Licenciatura, é incipiente e que tal lacuna é um entrave ao desenvolvimento tecnológico desses países.

5. Conclusões

Em função das recentes mudanças ocorridas na tecnologia de projeto de sistemas digitais, este trabalho, pretende a partir de uma discussão, contribuir para a reformulação e modernização do ensino de Eletrônica Digital em nossas Escolas de Engenharia e Tecnologia. Os futuros engenheiros precisam aprender como criar "sistemas em chip", programá-los e por meio da lógica reconfigurável repará-los. Pois esses dispositivos são os novos rumos das novas tecnologias de hardware para Automação Industrial.

Referências

[1] J. F. Wakerly Digital; "Design – Principles & Practices", Prentice Hall, ISBNO-13-769191-2, 3th Edition, New Jersey, Estados Unidos, 2000.

[2] R. J. Tocci; N. S. Widmer and G. L. K. Moss; "Digital System", Pearson Education International, 9 th ed., USA, 755p., 2004.

[3] M.A.Teixeira,"Técnicas de Reconfigurabilidade dos FPGAs da Família APEX 20K Altera". Tese de dissertação de mestrado, USP, São Carlos, 2002.

[4] C. Costa; "Projeto de Circuitos Digitais com FPGA", Editora Érica, ISBN 978-85-365-0239-7, 1.a edição, São Paulo, Brasil, 206p.,2009.

[5] I. Grout; "Digital System Design with FPGAs and CPLDs", Newnes, 1st ed.; Burlington, MA, USA, 784p., 2008.

[6] K. Coffman; "Real World FPGA design with Verilog", Prentice Hall PTR. ISBN 0-13-099851-6, New Jersey, Estados Unidos, 1999.

[7] S. Brow, Z. Vranesic; "Fundamentals of Digital Logic with VHDL, Design"*;* Mc Graw-Hill Series in Computer Enginnering, 2000.

Unidades ASH para paralelização de modelos acústicos DWM tridimensionais

Sara Barros Guilherme Campos
DETI / IEETA – Universidade de Aveiro
barros.s@ua.pt guilherme.campos@ua.pt

Resumo

Este artigo trata do desenvolvimento de uma rede dedicada de computação – o "Meshotron" – para paralelizar em larga escala modelos acústicos 3-D baseados em malhas de guias de onda digitais ('Digital Waveguide Meshes' – DWM). Descreve-se a arquitectura geral das unidades especializadas ('application-specific hardware' – ASH) que formarão esta rede e apresentam-se as etapas iniciais do projecto, a saber: desenvolvimento de um protótipo virtual através de ferramentas de simulação de hardware e implementação em FPGA de uma unidade de 'scattering' para a topologia de malha rectangular.

1. Modelação acústica DWM

A modelação por guias-de-onda digitais (*Digital Waveguides* – DW) é um método numérico de diferenças finitas no domínio do tempo (FDTD) para a resolução da equação de onda, baseado em discretização temporal e espacial [1]. Tem obtido grande sucesso, especialmente em síntese de som e simulação acústica de instrumentos. Por exemplo, os sintetizadores mais eficientes de instrumentos de cordas e sopro são baseados em modelos DW unidimensionais (1-D) [2].

É possível interligar DW em estruturas regulares, obtendo-se assim estruturas em malha (DWM) [3]. Os pontos de interligação denominam-se 'nós' ou 'junções' (*scattering junctions*). As malhas 2-D prestam-se, por exemplo, à simulação de instrumentos de percussão [4]. O caso 3-D é especialmente relevante em simulação acústica de salas [5][6][7].

2. Simulação de salas com DWM 3-D

O modelo DWM de uma sala consiste numa grelha de nós interligados por segmentos de DW unitários; o meio de propagação de som é discretizado em nós 'ar' e os materiais que o delimitam são representados por nós 'fronteira'. O número de vizinhos a que um nó genérico está ligado, n, é conhecido como *número de coordenação* da malha e depende da sua topologia; vale 4, 6, 8 e 12 para as malhas tetraédrica, rectangular, octaédrica e dodecaédrica, respectivamente [8].

Cada ciclo iterativo do algoritmo de modelação compõe-se de dois passos simples [3]. No primeiro (*scattering pass* – S), calcula-se o valor da variável de onda (e.g. pressão acústica, p) em cada nó a partir das componentes recebidas dos n nós vizinhos. Para nós de 'ar', considerando propagação sem perdas,

$$p = \frac{2}{n}\sum_{k=1}^{n} p_{k_{in}} \qquad \text{Eq. 1}$$

Em seguida, obtêm-se as componentes de pressão acústica a enviar a esses mesmos nós:

$$p_{k_{out}} = p - p_{k_{in}}, \quad k \in \{1,...,n\} \qquad \text{Eq. 2}$$

Note-se que, além de somas algébricas, os cálculos envolvem apenas uma divisão; se n for uma potência inteira de 2 (caso, em 3-D, das topologias tetraédrica e octaédrica) e o cálculo for realizado em formato inteiro, esta reduz-se a um simples *bit shift*.

Para os nós 'fronteira', pode adoptar-se a chamada terminação 1-D [5]; sendo R o coeficiente de reflexão acústica do material em causa,

$$p_{k_{out}} = R.p_{k_{in}}, \quad k \in \{1,...,n\} \qquad \text{Eq. 3}$$

O segundo passo (*delay pass* – D) consiste na transferência de dados entre nós, finda a qual é possível iniciar o *scattering pass* do ciclo seguinte.

Tomemos como referência a topologia 3-D mais simples – a rectangular. Como ilustra a Fig. 1, os seus nós modelam volumes cúbicos e interagem com cada um dos seus 6 vizinhos (designados L, R, B, F, D e U para evocar a respectiva posição: *left*, *right*, *back*, *front*, *down* e *up*) através de pares de registos de dados: um de recepção (*in*) e outro de envio (*out*).

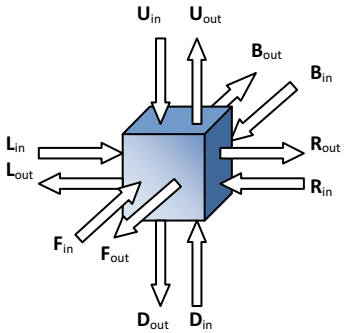


Fig. 1 – Nó genérico de uma DWM 3-D rectangular

Neste caso, os nós de 'ar' são regidos por:

$$\begin{cases} p = \dfrac{p_{L_{in}} + p_{R_{in}} + p_{B_{in}} + p_{F_{in}} + p_{D_{in}} + p_{U_{in}}}{3} & \text{Eq. 4} \\ p_{X_{out}} = p - p_{X_{out}}, \quad X \in \{L,R,B,F,D,U\} & \text{Eq. 5} \end{cases}$$

3. Tempo de cálculo da RIR

Dadas as posições de fonte e receptor, uma sala pode ser caracterizada acusticamente pela sua resposta impulsional (*Room Impulse Response* – RIR). Teoricamente, os modelos DWM 3-D permitem obtê-la de forma muito rigorosa, pois todos os fenómenos físicos associados à propagação de ondas (reflexão, absorção, difracção, interferência, …) são automaticamente tidos em conta. A principal dificuldade prática é a dimensão computacional do problema. Sendo t_{m1} o tempo necessário por nó para completar uma única iteração (*scattering* e *delay*), o tempo de cálculo para uma sala com volume V e tempo de reverberação[1] RT_{60} é dado por [8]

$$T_{RIR} = \frac{1}{\left(c\sqrt{3}\right)^3} V.RT_{60}.f_s^{\,4}.t_{m1} \qquad \text{Eq. 6}$$

onde c representa a velocidade de propagação do som (aproximadamente 344 m/s no ar em condições de ambiente normais) e f_s a frequência de amostragem, que está directamente relacionada com a densidade da malha; sendo d a distância entre nós,

$$f_s = \frac{c\sqrt{3}}{d} \qquad \text{Eq. 7}$$

Para ilustrar a dimensão do problema, considere-se uma sala de concerto com V=10000 m^3 e RT_{60}=1.5s – relativamente pequena e acusticamente "seca" [9]. Seja f_s=44.1 kHz (valor típico em áudio) – podem justificar-se taxas significativamente mais altas para combater o erro de dispersão, principal limitação intrínseca do método [10]. Tomando t_{m1}=50ns – optimista, a julgar pelos estudos de *benchmarking* efectuados em computadores de uso geral – o cálculo demoraria aproximadamente 155 dias [11]. Fica claro que, para que as aplicações práticas não se restrinjam a salas pequenas e/ou baixas frequências de amostragem, t_{m1} tem de diminuir drasticamente. É, por isso, indispensável paralelizar o modelo.

4. Computação paralela

Estudos anteriores exploraram *crowd computing* baseada em decomposição de dados: o modelo é dividido em blocos (de forma cúbica, uma vez que esta maximiza a granularidade) e um processo paralelo é associado a cada um deles. Esta estratégia, igualmente aplicável para qualquer topologia DWM 3-D (tetraédrica, rectangular, octaédrica ou dodecaédrica), foi testada em multi-processadores e *workstation clusters*. Confirmou-se que os modelos DWM se prestam muito bem a paralelização em grande escala, desde que a rede de computação paralela seja, ela própria, estruturada numa topologia de malha 3-D. Nestas condições, o *overhead* de comunicação e, por consequência, a eficiência da paralelização são independentes do número total de unidades que formam o modelo, n_p. Além disso, os testes indicaram que o *overhead* de comunicação se pode manter muito baixo, o que significa *speedup* sensivelmente proporcional a n_p; por exemplo, uma rede de 10x10x10 unidades permitiria diminuir t_{m1} por um factor próximo de 1000 [8][11][12].

5. O "Meshotron"

O objectivo deste projecto é explorar estas propriedades, criando unidades de *hardware* especificamente adaptadas para constituir uma rede de computação paralela de modelação DWM 3-D. Começou-se por considerar a topologia rectangular. Cada unidade modela uma partição cúbica, formada por N^3 nós, e inclui interfaces de comunicação, com respectivas unidades de controlo, para trocar dados com as unidades adjacentes (6, no caso geral)[2]. Cada interface é dotada de um *buffer* de envio e outro de recepção, com N^2 posições cada. Assim, com 12 *buffers* de comunicação, a estrutura de uma unidade replica, em maior escala, a de um nó (vide Fig. 1).

Ao contrário do *scattering pass*, que envolve apenas operações internas a cada nó (vide Eq. 1 a Eq. 3), o *delay pass* exige comunicação entre unidades, pois os nós situados nas faces de um bloco têm vizinhos nos blocos adjacentes. Designá-los-emos 'nós superficiais', por oposição àqueles cujos vizinhos pertencem todos à mesma partição – os 'nós interiores'. Em cada unidade, o *delay pass* é dividido nas seguintes fases:

D1. Os registos de envio dos nós superficiais destinados a unidades adjacentes são copiados para os respectivos *buffers* de envio.

D2. Os restantes registos de envio (de todos os nós) são copiados directamente para os registos de recepção correspondentes nos nós vizinhos (sempre na mesma unidade).

D3. Finalmente, os dados provenientes de unidades adjacentes, contidos nos 6 *buffers* de recepção, são copiados para os respectivos registos de recepção dos nós superficiais.

Dado que um nó possui 6 registos de envio, o total de registos enviados por iteração é $6N^3$. Desses, $6N^2$ destinam-se a unidades adjacentes, via *buffers* de envio. O *overhead* de comunicação (fases 1 e 3) pode assim ser estimado em $\frac{6N^2}{6N^3} = \frac{1}{N}$, tornando-se diminuto para dimensões razoáveis da partição cúbica (inferior a 2% para N=64, por exemplo). Deste modo, a sequência descrita permite assegurar que a troca de dados entre unidades não causa estados de espera no processamento, que quebrariam

[1] Intervalo de tempo necessário para que o campo sonoro sofra um determinada atenuação (60dB no caso de RT_{60}).

[2] Uma ligação adicional permitirá configurar o modelo a partir de um computador *host* e extrair resultados dos cálculos (RIR).

a eficiência da paralelização. De facto, a transferência de dados entre unidades (assegurada pelas interfaces de comunicação) pode decorrer imediatamente após a fase 1 e em simultâneo com a fase 2, de forma que os dados necessários na fase 3 se encontrem imediatamente disponíveis.

A vantagem do "Meshotron" não radica apenas no ganho inerente ao processamento paralelo com *communication overehad* muito baixo e independente do número de processadores. Sendo especificamente concebidas, as suas unidades poderão tirar o máximo partido da simplicidade do algoritmo de modelação DWM.

As vantagens potenciais do desenvolvimento de *hardware* especializado para modelação DW foram aliás apontadas desde o início pelos seus autores. Recentemente, foi concebido *hardware* baseado em FPGA para implementação de malhas 2-D usando paralelismo nodal; existe uma unidade por nó e a sua interligação física mapeia directamente a topologia da malha. Assim, todos os nós são processados em simultâneo [13]. Todavia, não parece possível transpor esta abordagem para o caso 3-D.

6. Arquitectura das unidades

A Fig. 2 esquematiza a arquitectura proposta. Cada nó requer 14 registos de dados: aos 6 de envio e 6 de recepção indicados na Fig. 1, acrescem um de configuração (*conf*), definindo o tipo do nó ('ar' ou 'fronteira' de dado material), e outro para guardar a pressão no nó (*p*). Por isso, o volume cúbico pode ser representado em 14 bancos de memória de N^3 posições; como sugere a figura, ao nó (*i*,*j*,*k*) correspondem as posições (*i*,*j*,*k*) em todos os bancos.

Na fase de *scattering* (S), o processamento é assegurado por blocos capazes de realizar os cálculos indicados na secção 2. São dotados de 14 portos: 7 de entrada, para obter os dados de configuração do nó (*conf*) e seus 6 registos de recepção; e 7 de saída, para fornecer a pressão acústica do nó (*p*) e seus 6 registos de envio.

A unidade deverá conter blocos de *scattering* em número suficiente para garantir que os dados podem ser processados à taxa máxima de acesso permitida pela memória. Isto implica um duplo varrimento da memória (leitura seguida de escrita) em troços sucessivos, correspondendo o comprimento destes troços ao número de blocos de *scattering*.

É necessário um 'gestor de barramentos' para impor as configurações e os esquemas de endereçamento adequados às 4 fases (S, D1, D2 e D3) de cada iteração. Na fase S, a gestão é particularmente simples, pois os dados a ler ou escrever pertencem a um único nó (mesmo endereço nos vários bancos).

A Tabela 1 rege o endereçamento no *delay pass*. Como ela evidencia, nas fases D1 e D3 a troca de dados envolve apenas nós superficiais e os *buffers* de interface. Basta, por isso, fixar uma das coordenadas e percorrer as restantes.

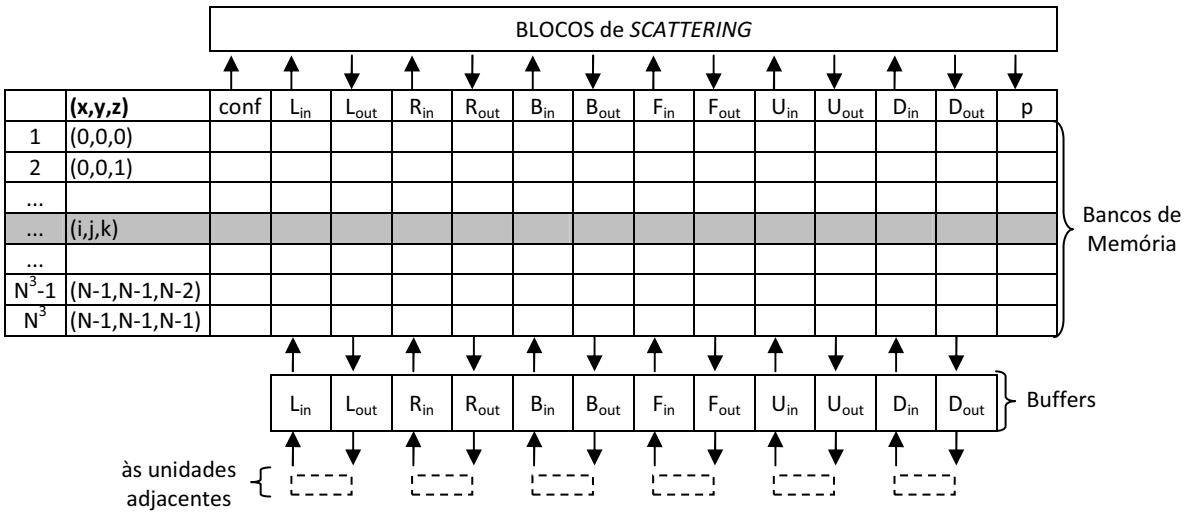


Fig. 2 – Esquema simplificado da arquitectura de uma unidade constituinte do *Meshotron*

D1:

	Origem		Destino
	Banco	Endereços	*Buffer*
Eixo x	L_{out}	(0,y,z)	L_{out}
	R_{out}	(N-1,y,z)	R_{out}
Eixo y	B_{out}	(x,0,z)	B_{out}
	F_{out}	(x,N-1,z)	F_{out}
Eixo z	D_{out}	(x,y,0)	D_{out}
	U_{out}	(x,y,N-1)	U_{out}

D2:

Origem			Destino	
Banco	Endereços		Banco	Endereços
L_{out}	(x,y,z)	x>0	R_{in}	(x-1,y,z)
R_{out}	(x,y,z)	x<N-1	L_{in}	(x+1,y,z)
B_{out}	(x,y,z)	y>0	F_{in}	(x,y-1,z)
F_{out}	(x,y,z)	y<N-1	B_{in}	(x,y+1,z)
D_{out}	(x,y,z)	z>0	U_{in}	(x,y,z-1)
U_{out}	(x,y,z)	z<N-1	D_{in}	(x,y,z+1)

D3:

Origem	Destino	
Buffer	Banco	Endereços
L_{in}	L_{in}	(0,y,z)
R_{in}	R_{in}	(N-1,y,z)
B_{in}	B_{in}	(x,0,z)
F_{in}	F_{in}	(x,N-1,z)
D_{in}	D_{in}	(x,y,0)
U_{in}	U_{in}	(x,y,N-1)

Tabela 1 – Endereçamentos nas sucessivas fases do *delay pass* (D1, D2 e D3)

Note-se como o facto de o processamento relativo aos nós superficiais ser parcialmente realizado nas fases D1 e D3 evita qualquer problema de detecção de limites no endereçamento dos nós vizinhos na fase D2.
Sublinhe-se que esta arquitectura permite notável paralelismo interno em todas as fases do *delay pass*, pois as transferências são executadas em simultâneo em todos os 6 sentidos (L, R, B, F, D e U).
A troca de dados entre *buffers* e unidades adjacentes é assegurada por unidades de comunicação independentes operando durante a fase D2.

7. Trabalho Futuro

O projecto prosseguirá com a implementação do modelo ilustrado na Fig. 2 recorrendo a ferramentas de simulação de *hardware*. Tal permitirá avaliar o comportamento global do sistema, simular e escolher alternativas de implementação e estabelecer uma plataforma de integração e teste dos blocos de *hardware* a desenvolver.

Em simultâneo, iniciar-se-á a implementação em FPGA do bloco de *scattering*, considerando apenas nós de 'ar' (Eq. 4 e Eq. 5), numa primeira abordagem. Investigar-se-ão as possibilidades de implementação combinacional e sequencial. Seguir-se-á a generalização para nós 'fronteira', que, na abordagem de terminação 1-D (Eq. 3), não envolvem acréscimo de complexidade.

8. References

[1] Smith III, J O (1992) "Physical Modeling Using Waveguides," *Computer Music Journal*, vol. 16(4), Winter, pp. 74-91.

[2] Smith III, J O (1998) "Principles of Digital Waveguide Models of Musical Instruments," in M Kahrs and K Brandenburg (eds.), *Applications of Digital Signal Processing to Audio and Acoustics*, pp. 417-466. Kluwer Academic Publishers.

[3] Van Duyne, S and Smith III, J O (1993) "Physical Modeling with the 2-D Digital Waveguide Mesh," *Proc. Int. Computer Music Conf. (ICMC'93)*, Tokyo, Sept., pp. 40-47.

[4] Fontana, F and Rocchesso, D (1998) "Physical Modelling of Membranes for Percussion Instruments," *Acustica – Acta Acustica*, vol. 84, May/June, pp. 529-542.

[5] Savioja, L; Karjalainen, M and Takala, T (1996) "DSP Formulation of a Finite Difference Method for Room Acoustics Simulation," *Proc. IEEE Nordic Signal Processing Symp. (NORSIG'96)*, Espoo, Finland, 24-27 Sept., pp. 455-458.

[6] Van Duyne, S and Smith III, J O (1996) "The 3-D Tetrahedral Digital Waveguide Mesh with Musical Applications," *Proc. Int. Computer Music Conf. (ICMC'96)*, Hong-Kong, Aug., pp. 9-16.

[7] Campos, G, Howard, D M and Dobson, S (2001) "Acoustic Reconstruction of Music Performance Spaces using Three-Dimensional Digital Waveguide Mesh Models," *Proc. Int. Symp. on Musical Acoustics (ISMA'2001) – Musical Sounds from Past Millenia*, Perugia, Italy, 10-14 Sept., pp. 581-584.

[8] Campos, G (2003) *Three-Dimensional Digital Waveguide Mesh Modelling for Room Acoustic Simulation*. Ph.D. thesis. University of York.

[9] Beranek, L (1996) *Concert and Opera Halls: How They Sound*. Woodbury, New York: Acoustical Society of America.

[10] Campos, G and Howard, D M (2005) "On the Computational Efficiency of Different Waveguide Mesh Topologies for Room Acoustic Simulation," *IEEE Trans. Speech Audio Process.*, vol. 13(5), Sept., pp. 1063-1072.

[11] Campos, G and Howard, D M (2000) "On the Computation Time of Three-Dimensional Digital Waveguide Acoustic Models," *Proc. 26th Euromicro Conf.*, Maastricht, Holland, 5-7 Sept., vol. II, pp. 332-339.

[12] Campos, G and Howard, D M (2000) "A Parallel 3D Digital Waveguide Mesh Model with Tetrahedral Topology for Room Acoustic Simulation," *Proc. COST G-6 Conf. on Digital Audio Effects (DAFx-00)*, Verona, Italy, 7-9 Dec., pp. 73-78.

[13] Motuk, E; Woods, R and Bilbao, S (2005) "FPGA-based Hardware for Physical Modelling Sound Synthesis by Finite Difference Schemes," *IEEE Int. Conf. Field-Programmable Technology (FPT'05)*, Singapore, 11-14 Dec., pp. 103-110.

Índice de Autores

Notas