

Arithmetic Soft-Core Accelerators

On the cover: The Kalasasaya Temple in Tiwanaku (presented in the photograph) is a ritual platform with a size of 130m by 120m. The walls were made of huge blocks of red sandstone and andesite. The blocks are precisely fitted to form a platform base 3m high. The massive entrance steps are flanked by two monolithic uprights. Also a partial view of the semi-subterranean temple is appreciated with some sculptural heads in its walls. The city of Tiwanaku (also spelled Tiahuanaco), located in La Paz - Bolivia, was the capital of the “Aymara civilization” that dominated the Andean region between 500 and 900 A.D.

Arithmetic Soft-Core Accelerators

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op dinsdag 27 november 2007 om 10:00 uur

door

Daniel Ramiro Humberto **CALDERON ROCABADO**

Electrical Engineer, M.Sc. Computer Sciences
Instituto Tecnológico de Costa-Rica
M.Sc. Modern Control Systems
Universidad Mayor de San Simón
geboren te La Paz, Bolivia

Dit proefschrift is goedgekeurd door de promotoren:

Prof. dr. K.G.W. Goossens

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter	Technische Universiteit Delft
Prof. dr. K.G.W. Goossens, promotor	Technische Universiteit Delft
Prof. dr. A.V. Veidenbaum	University of California - Irvine
Prof. dr. J. Takala	Tampere University of Technology
Prof. dr. ir. A.J. van der Veen	Technische Universiteit Delft
Prof. dr. P.J. French	Technische Universiteit Delft
dr. A.D. Pimentel	Universiteit Amsterdam

This thesis would never be completed without the scientific guidance and inspiration of Stamatis Vassiliadis.

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Calderón R., Daniel Ramiro Humberto
Arithmetic Soft-Core Accelerators
Daniel Ramiro Humberto Calderón Rocabado. – [S.l. : s.n.]. – Ill.
Thesis Technische Universiteit Delft. – With ref. –
Met samenvatting in het Nederlands.

ISBN 978-90-807957-7-8

Subject headings: adaptable machines, hardware accelerators, media processing, performance, prototyping.

Copyright © 2007 Daniel Ramiro Humberto CALDERON Rocabado
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without permission of the author.

Printed in The Netherlands

*To my family
with infinite gratitude and love*

Arithmetic Soft-Core Accelerators

Daniel Ramiro Humberto Calderón Rocabado

Abstract

In this dissertation, we address the design of multi-functional arithmetic units working with the most common fixed-point number representations, namely: unsigned, sign-magnitude, fractional, ten's and two's complement notations. The main design goal is to collapse multiple complex arithmetic operations into a single, universal arithmetic unit, aiming at the highest possible reutilization of shared hardware resources. More specifically, we propose an Arithmetic unit for collapsed Sum-of-Absolute Differences (SAD) and Multiplication operations (AUSM). This unit collapses various multi-operand addition based operations, such as SAD, universal notation multiplication, Multiply-Accumulate (MAC), fractional multiplication. Our AUSM design demonstrated high hardware reutilization level of up to 75%, yet its performance is comparable to the fastest related stand-alone designs supporting the individual operations. Another complex arithmetic operation, considered in this thesis is Matrix-Vector Multiplication. We collapsed fixed-point dense and sparse matrix-vector multiplication in one unit. It's Xilinx Virtex II Pro implementation suggests up to 21GOPS on a xc2vp100-6 FPGA device. Furthermore, in this thesis, we propose an arithmetic unit for universal addition, which supports addition/subtraction in binary and Binary Coded Decimal (BCD) representations in various sign notations. The hardware reutilization level for this unit was 40% and its performance was estimated to be more than 82MOPS. All considered units require massively parallel memory organizations, capable of providing high data throughput. Therefore, in this thesis, we propose a high-performance address generator (AGEN) employing a modified version of the low-order interleaved memory access approach. Our experiments suggest that the AGEN can produce 8×32 - bit addresses every 6 ns. Overall, in this dissertation, we demonstrated a design approach, which allows collapsing of multiple arithmetic operations into performance efficient universal designs with high level of hardware reutilization among the implemented functions.

Acknowledgements

The work presented in this dissertation has been carried out at the Computer Engineering (CE) Laboratory, division within the Electrical Engineering, Mathematics and Computer Science (EEMCS) faculty at Delft University of Technology (TU Delft). During the last four years I came across many people who supported me and assisted me. I would like to take the opportunity to thank them.

My foremost acknowledgements go to my advisor Professor Dr. Stamatis Vassiliadis, chair of the CE Laboratory at TU Delft, for giving me the opportunity to work with him after a short personal interview. He gave me the opportunity to learn this amazing field of the Computer Engineering and particularly the Adaptable Computing field. Regrettably, he return so early, leaving an incommensurable hole in the Computer Architecture community and a bitter taste in the people that have had the opportunity to know him.

I acknowledge to Prof. dr. Kess Goossens, my promotor and new member at the CE-Group. His guidance in the set up of the final thesis defense was important to conclude my work at TU Delft.

I am equally indebted to Dr. Georgi Gaydadjiev, who have been my mentor since Stamatis has gone. His suggestions and attention for finalizing the dissertation was invaluable as well as the help of Dr. Georgi Kuzmanov.

My special tanks to Egbert Bol, he showed me the path to reach this beautiful town called Delft, and introduced me in Professor Vassiliadis group. My family and me will remember you as well as the friendship with MSc. Jing Miao.

I want to thank all colleagues at the CE lab for providing a fun and friendly place to work in, specially to Dr. Arjan van Genderen. The Italians, Greeks, Iranians, Pakistanis, Bulgarians, Romanians, Indians, Nepalis and South Americans make a great environment where I have been able to learn so much of different cultures. Thanks to Bert Meijs and Lidwina Tromp for all your support.

To date, I also consider this dissertation an emanation of my lifetime study. Therefore, I would like to express gratitude to all my teachers including Prof. Dr. Ger Honderd and Dr. Wim Jongkind, who greatly contributed for building my background of knowledge.

Finally, with my deepest love and gratitude, I would like to thank my parents, Dr. Humberto Calderón Manrique and Betty Rocabado Mercado, for their love, patience, trust, advices, and support during the entire life of mine. Last but not least I want to thank my wife Roxana Antezana A. and my son Raniero Humberto Calderón A. for being more than understanding when I was just working and working and frequently, bringing my work problems at home.

D.R.H. Calderón R.

Delft, The Netherlands, 2007

Contents

Abstract	i
Acknowledgments	iii
List of Tables	ix
List of Figures	xi
List of Acronyms	xiv
1 Introduction	1
1.1 Problem Overview	4
1.2 Research Questions	5
1.3 Dissertation Overview	6
2 Arithmetic Unit for collapsed SAD and Multiplication operations (AUSM)	11
2.1 Unit Collapsing Example	12
2.2 The AUSM Array Organization	17
2.3 Experimental Results	24
2.4 Conclusions	27
3 AUSM extension	29
3.1 The AUSM Extension: a Multiple-Addition Array	30
3.1.1 Previous Work	30

3.1.2	Array Sectioning	31
3.1.3	Array Organization	33
3.2	The Array Construction - Equations Description	37
3.3	Experimental Results and Analysis	47
3.4	Conclusions	48
4	Fixed Point Dense and Sparse Matrix-Vector Multiply Arithmetic Unit	51
4.1	Background	52
4.1.1	Sparse Matrix Compression Formats	53
4.2	Dense and Sparse Matrix-Vector Multiply Unit	57
4.2.1	The Pipeline Structure	59
4.2.2	Sparse Matrix-Vector Multiply Example	64
4.2.3	Reconfigurable Optimizations:	66
4.3	Experimental Results	67
4.4	Conclusions	70
5	Arithmetic Unit for Universal Addition	71
5.1	Background	72
5.1.1	Related Work	73
5.2	Reconfigurable Universal Adder	76
5.2.1	Decimal Arithmetic Additions	79
5.2.2	Decimal Subtraction Example	82
5.3	Experimental Results Analysis	83
5.4	Conclusions	84
6	Address Generator for complex operation Arithmetic Units	87
6.1	Background	88
6.2	AGEN Unit Design	90
6.2.1	Memory-Interleaving Mechanism	92
6.2.2	The AGEN Design	96
6.3	Experimental Results Analysis	99

6.4	Conclusions	100
7	Comparative Evaluations	103
7.1	Arithmetic Accelerators	103
7.1.1	The SAD Accelerator	103
7.1.2	The Dense and Sparse Matrix-Vector Multiply Unit . .	106
7.1.3	The Binary and Decimal Adder	108
7.2	System Integration of the SAD Unit	109
7.3	Hardware Reutilization	113
7.3.1	Architectural Adaptability	116
7.4	Conclusions	118
8	General Conclusions	119
8.1	Summary	119
8.2	Contributions	121
8.3	Advantages of the Collapsed Arithmetic Hardware	123
8.4	Proposed Research Directions	124
A	Reconfigurable architectures survey	127
A.0.1	A General-Purpose Programmable Video Signal Processor (VSP)	127
A.0.2	Processor Reconfiguration through Instruction-Set Metamorphosis (PRISM)	128
A.0.3	Reconfigurable Architecture Workstation (RAW) . . .	129
A.0.4	PipeRench	129
A.0.5	Garp: a MIPS Processor with a Reconfigurable Co-processor	130
A.0.6	MorphoSys: a Coarse Grain Reconfigurable Architecture	131
A.0.7	Matrix: a Coarse grain Reconfigurable Architecture . .	132
A.0.8	Architecture Design of Reconfigurable Pipelined Data-paths (RaPiDs)	133
A.0.9	ADRES	134

A.0.10 MOLEN: a customizable adaptable architecture	136
Bibliography	137
List of Publications	153
Samenvatting	155
Curriculum Vitae	157

List of Tables

1.1	Multiple-addition related kernels	3
2.1	Universal multiplier extensions - MSB use in addition operation	19
2.2	Selector signal SEL1 and SEL2 - Multiplexer behavior	21
2.3	(3:2)counters used in the AUSM scheme	22
2.4	AUSM and other multiplication units.	25
2.5	AUSM and other multiplication units	26
3.1	Input and output of the AUSM extension array	32
3.2	Selector signal SEL3 - Multiplexer behavior	36
3.3	$i1_{(j,i)}$ inputs: The 8-to-1 multiplexer	37
3.4	AUSM extension and related units	48
3.5	AUSM extension and related units	49
4.1	Comparison of matrix-vector formats	57
4.2	Resources allocation control bits.	62
4.3	Matrix-vector multiply/add unit.	68
4.4	Routing multiplexers - Second Stage.	68
4.5	Allocation hardware -Third stage.	69
4.6	Matrix-vector multiplication unit.	69
5.1	Adders - data representation	76
5.2	Adder setting up considerations	77
5.3	Decimal digit correction terms	80

5.4	Latency&area - Adder comparison	84
6.1	Correspondence $a_\alpha \leftrightarrow (i_\alpha, j_\alpha) \leftrightarrow a_{\alpha n} \leftrightarrow a_{\alpha 10}$ for $n = 8$ and Str = 3.	94
6.2	Hardwired encoder - Set up table of equations	99
6.3	The address generation unit and embedded arithmetic units . .	100
7.1	Area and latency used to process one CIF	105
7.2	Dense matrix multiplication comparison - Related work	107
7.3	Microblaze based SoC (@ 100 MHz)	109
7.4	Vector coprocessor: SAD case	110
7.5	Software - hardware comparison	112
7.6	Hardware reuse - AUSM extended	114
7.7	Common hardware blocks - Arithmetic soft-core accelerators .	115
7.8	Comparison of adaptable processor features	117

List of Figures

2.1	Ripple Carry Adder (RCA)	12
2.2	Basic array multiplier unit	12
2.3	Two's complement sign extension scheme and example	13
2.4	SAD (3:2)counter array	15
2.5	Common basic array	16
2.6	Cell detail in Figure 2.5	16
2.7	The AUSM: a 16x16 integer multiplication and SAD operation.	17
2.8	(3:2)counters of the AUSM array	22
3.1	Multiple operation units: a) AUSM array b) AUSM extended array	31
3.2	The AUSM extension array scheme	31
3.3	Fractional multiplication example.	34
3.4	Detail organization into the AUSM extended array	36
3.5	The AUSM extension - array organization	38
4.1	Sparse matrix representation	53
4.2	Compressed Row Storage (CRS) format	54
4.3	BBCS format	54
4.4	BBCS format example	55
4.5	HiSM format example	56
4.6	Vector read	59
4.7	Multiplication reduction trees	60

4.8	Reduction trees (multiple-addition)	61
4.9	Final addition	65
4.10	Sparse by dense vector multiplication for s=4: a) First processing step, b)Second processing step.	65
4.11	Scalability on dense matrix processing	66
5.1	Decimal subtraction	73
5.2	Hwang's proposal [1]	74
5.3	Fischer's proposal [2]	74
5.4	Haller's proposal (z900)	75
5.5	Sign magnitude adder	78
5.6	Decimal subtraction: $N_2 > N_1$	79
5.7	Universal adder micro-architecture [3]	81
5.8	Decimal subtraction: double addition scheme	82
5.9	Decimal subtraction: collapse approach	83
6.1	Interleaved memory formats.	88
6.2	Main accumulator circuitry	89
6.3	Interleaved memory pipelined access to memory	90
6.4	Block diagram of the reconfigurable custom computing unit [4]	91
6.5	An 8-way interleaved memory banks with odd strides ≤ 15	93
6.6	Compound instruction	96
6.7	Address generation unit: (a) Accumulator for BS computing, (b) Accumulator for loop control, (c) Hardwired encoder, (d) Index accumulator, (e) Final addition effective address computing	97
6.8	Main accumulator circuitry	98
7.1	Universal Adder - Speed up characteristics	109
7.2	GPP and vector coprocessor interface - Fast simplex link dedicated channel	111
7.3	SAD processing: (a) Regular approach. (b) Our approach with unit collapsing	112

7.4	Arithmetic soft-cores accelerators: (a) AUSM [5], (b) AUSM extension [6].	113
7.5	Arithmetic accelerators: a) Dense Sparse Matrix Vector Multiply Unit b) Universal Adder	114
A.1	Contents of the ALE [7]	128
A.2	PRISM II Hardware Platform [8]	129
A.3	RAW architecture [9]	130
A.4	PipeRench architecture [10]	130
A.5	Garp architecture [11]	131
A.6	MorphoSys architecture [12]	132
A.7	Matrix architecture [13]	133
A.8	RaPid architecture [14]	134
A.9	Adres architecture [15]	135
A.10	MOLEN machine organization [16]	136

List of Acronyms

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Integrated Processor
AUSM	Arithmetic Unit for collapsed SAD and Multiplication Operations
BBCS	Block Based Compression Storage
BCD	Binary Coded Decimal
CIF	Common Intermediate Format
CCU	Custom Computing (Configurable) Unit
CRS	Complex Row Storage
DMVM	Dense Matrix-Vector Multiplication
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
HiSM	Hierarchical Sparse Matrix format
HDL	Hardware Description Language
HDTV	High-Definition Television
HW	Hardware
IIR	Infinite Impulse Response
ISA	Instruction Set Architecture
MAC	Multiplication and Accumulation
ME	Motion Estimation (in all MPEGs)
MPEG	Motion Pictures Experts Group
NZE	Non Zero Elements
PE	Processing Element
PELs	Picture elements
PDA	Personal Digital Assistants
RP	Reconfigurable Processor
SAD	Sum of Absolute Differences
SoC	System on a Chip
SMVM	Sparse Matrix-Vector Multiplication
NoC	Networks-on-Chip

Chapter 1

Introduction

Various emerging applications in the embedded and general purpose processing domains are envisioned to benefit from high performance arithmetic support. High definition streaming video and 3D graphics for mobile devices, pervasive gaming [17, 18] and financial transactions [19, 20] are just some examples of such applications. In addition, embedded devices like personal digital assistants (PDAs), handhelds, next generation mobile phones [21] and even interactive high definition TV (HDTV) sets are expected to support increasing number of additional functionalities in the near future. Additional computational problem is imposed by new applications with dynamically changing performance requirements. For example, in the new open framework for multimedia applications MPEG21 [22] support for wide range, dynamically selectable, audio and video resolutions is required [22, 23]. Furthermore, all classical issues such as silicon area and low power consumption are expected to grow in importance.

The current industrial solution to cope with the above trends is by using highly specialized processing elements for each new functionality class. This leads to very complex, heterogeneous systems with unacceptable development times and high associated costs. Multi-core processing platforms, originally proposed to address the state-of-the-art technology limitations, might be considered as a potential solution for some of the above problems [24, 25]. It should be noted that many additional shortcomings, e.g., communication and area overheads due to the interconnecting network and under-utilization of the distributed resources, are inherent to such platforms. Using closely coupled arithmetic accelerators is envisioned as a valid alternative that does not intro-

duce significant limitations. The application specific accelerators approach, widely used in the general purpose processors (GPP) domain [26, 27], is considered not feasible for many embedded devices due its hardware inflexibility and highly targeted customization, expressed in the basic instructions selection. Customized, adaptable processing units supporting complex operations rather than simple instructions (also referred as hardware accelerators) that specifically target the computational bottlenecks of applications, such as the one mentioned above, are investigated in this dissertation. More precisely, high-performance, silicon area efficient, parameterizable arithmetic units with multiple operations support are discussed in detail. Our units built upon elements based on arithmetic addition allow us to reuse most of the hardware resources among different operations types.

Four motivating examples that indicate the required performance in different cases are listed below:

- Motion estimation kernels [28] in multimedia processing, e.g., the Sum of Absolute Difference (SAD) for 16×16 picture element (pels) blocks would need 256 absolute subtract operations and 255 additions to compute one of the 255 motion vectors [29]. The computation of all 225 candidate motion vectors when video-streams of 352×288 pels frames with 30 frames per second (fps) is considered would require 1.365 GOPS (Giga operations per second);
- Matrix by vector multiplication, heavily used in game simulation [30], will require n multiplications and $n - 1$ additions per final dot product when matrix and vector sizes of $m \times n$ and respectively n are considered. The multiplication of a square matrix with $m = n = 1200$ by a vector of length $n = 1200$ needs 1.726 GOPS to produce results every second;
- Fixed point multiplication and multiply-add operations in many signal processing kernels [31, 32], e.g., a simple FIR filter will require N multiplications and $N - 1$ additions. Considering an audio application with a sample rate of 20 KHz and a 6 tap FIR filter will require 0.25 MOPS;
- BCD arithmetic used in financial operations; e.g. when a simple BCD addition is performed in software, it will require between 100 to 1000 longer times compared to hardware acceleration [33, 34].

The arithmetic operations involved on the above kernels together with some additional ones are presented in Table 1.1.

Table 1.1: Multiple-addition related kernels

Kernel	Equation
SAD	$\sum_{j=1}^{16} \sum_{i=1}^{16} A(x+i, y+j) - B((x+r)+i, (y+s)+j) $
Sum of Differences	$\sum_{j=1}^{16} \sum_{i=1}^{16} (A(x+i, y+j) - B((x+r)+i, (y+s)+j))$
Dense/Sparse Matrix-Vector Multiply	$\vec{c} = \vec{A} \times \vec{b}$ with $c_i = \sum_{k=0}^{n-1} A_{(i,k)} x b_k$
BCD Addition/Subtraction	$SUM = A + B + 10 + 6 + 6 \dagger$
Finite Impulse Response Filters	$y_i = \sum_{j=0}^{k-1} C_j X_{i-j}$
Infinite Impulse Response Filters	$y_i = \sum_{j=0}^{k-1} C_j X_{i-j} - \sum_{j=1}^{k-1} C_j Y_{i-j}$
2D Discrete Cosine Transform	$F_{(u,v)} = \frac{C_u C_v}{4} \sum_{j=0}^7 \sum_{i=0}^7 f(x,y) \frac{\cos((2x+1)u\pi)}{16} \frac{\cos((2y+1)v\pi)}{16}$
2D Inverse Discrete Cosine Transform	$f_{(x,y)} = \frac{C_u}{2} \sum_{j=0}^7 \left[\frac{C_v}{2} \sum_{i=0}^7 F(u,v) \frac{\cos((2x+1)u\pi)}{16} \right] \frac{\cos((2y+1)v\pi)}{16}$
Illumination Model	$I = k_a I_a + k_d I_l(\vec{N} \times \vec{L}) + k_s I_l(\vec{N} \times \vec{H})^{Ns}$

† : The first “10” and “6” values are added conditionally, the addition depends on the type of operation and the decimal Carry out (Cout) occurrence. Furthermore, the final “6” value is added when $B > A$ in subtraction operations when the post complement operation is required.

The following similarities among these arithmetic operations were found:

1. All kernels require multiple addition operations. They can be embedded in a loop as in the SAD case or be required just once as in the BCD and the illumination model cases. Some of the kernels implicitly require additions of the partial products from different operations, e.g., the dense and the sparse matrix multiplication and FIR.
2. Some kernels, e.g. FIR and IIR, involve dot product operations embedded into an outer loop performing additions that can benefit from a multiply-accumulate support. The Discrete Cosine transform can be also considered as such kernel when the cosine functions are implemented as look up tables.
3. Some of the kernels contain data dependent operations (not shown in detail in the table). Arithmetic comparison operations support can be

useful to hide the penalties of such operations. For example, in the BCD subtraction case, the three correcting additions required originally (one permanent and two conditional) can be reduced to only two. Similar optimizations can be applied to the SAD kernel.

All operations above, including the comparison operation, are based on (multiple) arithmetic additions. Based on this observation, we envision that complex, multiple domain arithmetic accelerators that effectively reuse the majority of the hardware resources are possible by using addition based building blocks. Having said this, those accelerators should support further customizations highly dependent on the specific kernel properties.

The remainder of this chapter is organized as follows. Section 1.1 briefly discusses the traditional solutions for the stated problem, and presents the background for our new proposal. Section 1.2 presents the open questions and methodology. This chapter is concluded in Section 1.3 that presents the research goal and overview of the dissertation.

1.1 Problem Overview

With the current technology, the widely used library-based approach shows a behavior that is heavily influenced by the wire delays [35]. Typically, multiple arithmetic blocks are interconnected to implement complex operations. Since the optimization scope of the arithmetic functional blocks is limited to their boundaries only, the implementation of complex operations with low latencies is becoming more and more difficult. In this thesis, we explore the benefits of breaking the boundaries of the aforementioned standard arithmetic blocks by collapsing them into fewer, but more complex structures. These structures have shorter latencies when compared to their interconnected library-based design counterparts.

The trend that we develop further in this thesis was inspired by some earlier Computer Arithmetic work. One of the pioneering approaches was introduced by Swartzlander in his *Merged Arithmetic* [36] proposal. In this design, the boundaries between multiplication and addition are dissolved when a multiply-accumulate operation is performed. This approach is nowadays widely used in DSP processors [37]. This concept was improved in a later proposal, *Compound Arithmetic*, where several arithmetic operations are

supported by the same functional unit [38]. Compound arithmetic relies on the fact that specific sequences of simple arithmetic operators like addition, subtraction, multiplication and division are present in many computer programs. Merging such sequences together can be exploited to improve performance. Good examples are the “*Fast Computation of Compound Expressions in Two’s Complement Notation*” [39] and the “*High-performance 3-1 Interlock Collapsing ALU’s*” [40]. The collapsing of several operations into a single hardware unit and optimizing the delay is the approach that we explore in this dissertation. We propose a set of arithmetic accelerators that extend the paradigm established by the aforementioned proposals. More precisely, our accelerators collapse up to 16 arithmetic functions in highly optimized hardware structures. The key feature of our designs are reduced latencies. In addition, we always consider optimal reuse of hardware resources. To validate our claim, we will use the widely available reconfigurable Field Programmable Gate Array (FPGA) technology and relate our approach to the previous art. However, this does not limit the implementation possibilities for the proposed approach. The same methodology is directly applicable to ASIC design processes.

1.2 Research Questions

In this thesis, we address performance efficient designs of arithmetic units, which employ common general purpose operations such as addition, subtraction, and multiplication, but also support domain specific operations such as sum-of-absolute differences and matrix-vector multiplications. In our work, we consider the most popular fixed-point number representations, namely two’s complement, ten’s complement, unsigned, and sign-magnitude notations. Our main idea is to exploit the functional commonalities between different arithmetic operations and to reuse hardware resources in order to obtain efficient designs of complex arithmetic operations. Pursuing this idea, we state four general research questions, which are addressed in this thesis. Answered chronologically, these questions led us to the efficient design of several complex arithmetic units, which are presented further in this dissertation. We can shortly state these question as follows:

1. **Can we identify a particular set of arithmetic operations, which share common arithmetic kernels?**

After identifying such a set of arithmetic operations and the corresponding set of common arithmetic kernels, a second main question drives our research:

2. **Can we collapse the identified arithmetic operations into a single complex arithmetic unit by sharing common hardware resources?**

In this thesis, we suggest several designs, which employ common hardware resources to support different arithmetic operations. For example, we merge in one unit the following operations: multiply, SAD, Multiply and Accumulate (MAC) - all of them in different number notations. The complete list of units, considered in this thesis is presented in Section 1.3. Our third research question is:

3. **What are the design advantages of such collapsed operations compared to traditional implementations with interconnected standard units?**

We synthesized our designs for reconfigurable technology. The synthesis results suggest that collapsing several operations into one unit allows shorter critical paths reduction compared to the traditional multiple-unit approaches. Finally, we investigate the complete design answering the last research question:

4. **Can we easily extract and implement sub-circuits based on our complex accelerators without having to redesign the entire unit?**

The partitioning of our designs allows the collapsed functionalities to be extracted in separate self-contained designs for individual use. Synthesis results suggest improved critical paths, for designs supporting such individual operations.

1.3 Dissertation Overview

The further discussion in this dissertation is organized as follows:

In Chapter 2 titled “**Arithmetic Unit for collapsed SAD and Multiplication operations (AUSM)**”, we present a configurable unit that collapses various

multi-operand addition related operations into a single array. Specifically we consider multiplication and sum of absolute differences and propose an array of processing elements capable of performing the aforementioned operations for unsigned, signed magnitude, and two's complement representations. The proposed array is divided into common and controlled logic blocks intended to be reconfigured dynamically. The proposed unit was constructed around three main operational fields, which are fed with the necessary data products or SAD addition terms in order to perform the desired operation. It is estimated that 66.6 % of the (3:2)counter array is shared by the operations providing an opportunity to reduce the reconfiguration times. The synthesis result for a FPGA device, of the new structure, is compared against other multiplier organizations. The obtained results indicate that the proposed unit is capable of processing 16 bit multiplication in 23.9 ns, and that an 8 input SAD can be computed in 29.8 ns when targeting Virtex II Pro-6 FPGA technology. Even though the proposed structure incorporates more operations, the additional delay when compared to conventional structures is negligible (in the order of 1% compared to Baugh&Wooley multiplier).

In Chapter 3 titled “**AUSM extension**”, we extend the functionality of the array presented in the previous chapter, augmenting more functionalities and preserving the original ones. A universal array unit is used for collapsing eight multi-operand addition related operations into a single and common (3:2)counter array. We consider for this unit multiplication in integer and fractional representations, the sum of absolute differences in unsigned, signed magnitude and two's complement notations. Furthermore, the unit also incorporates a multiply-accumulation unit for two's complement representation. The proposed multiple operation unit was constructed around 10 element arrays that can be reduced using well known counter techniques, which are feed with the necessary data to perform the proposed eight operations. It is estimated that 6/8 of the basic (3:2)counter array is re-used by different operations. The obtained results of the presented unit indicate that it is capable of processing a 4×4 SAD macro-block in 36.35 ns and it takes 30.43 ns to process the rest of the operations using a VIRTEX II PRO xc2vp100-7 FPGA device.

Chapter 4, titled “**Fixed Point Dense and Sparse Matrix-Vector Multiply Arithmetic Unit**”, presents a reconfigurable hardware accelerator for processing fixed-point-matrix-vector-multiply/add operations. The unit supports dense and sparse matrices in several well known formats. The prototyped

hardware unit accommodates 4 dense or sparse matrix inputs and performs computations in a space parallel design achieving 4 multiplications and up to 12 additions at 120 MHz over an xc2vp100-6 FPGA device, reaching a throughput of 1.9 GOPS. A total of 11 units can be integrated in the same FPGA chip, achieving a peak performance of 21 GOPS.

Chapter 5, titled “**Arithmetic unit for Universal Addition**”, presents an adder/subtractor arithmetic unit that combines Binary and Binary Code Decimal (BCD) operations. The proposed unit uses effective addition/subtraction operations on unsigned, sign-magnitude, and various complement representations. Our design overcomes the limitations of previously reported approaches that produce some of the results in complement representation when operating on sign-magnitude numbers. When reconfigurable technology is considered, a preliminary estimation indicates that 40 % of the hardware resources are shared by the different operations. This makes the proposed unit highly suitable for reconfigurable platforms with partial reconfiguration support. The proposed design, together with some classical adder organizations, were compared after synthesis targeting 4vfx60ff672-12 Xilinx Virtex 4 FPGA. Our design achieves a throughput of 82.6 MOPS with almost equivalent area-time product when compared to the other proposals.

In Chapter 6, titled “**Address Generator for Arithmetic Units with multiple complex operations**”, we describe an efficient data fetch circuitry for retrieving several operands from an 8-bank interleaved memory system in a single machine cycle. The proposed address generation (AGEN) unit operates with a modified version of the low-order-interleaved memory access approach. Our design supports data structures with arbitrary lengths and different (odd) strides. A detailed discussion of the 32-bit AGEN design aimed at multiple-operand functional units is presented. The experimental results indicate that our AGEN can produce 8 x 32-bit addresses every 6 ns for different stride cases when implemented on VIRTEX-II PRO xc2vp30-7ff1696 FPGA device using trivial hardware resources.

In Chapter 7, titled “**Comparative Evaluations**”, we present a comparison of the proposed arithmetic accelerator units with related work, in terms of occupied area and performance. We present the results of mapping a complete vector-coprocessor micro-architecture for the SAD case; we compare the results of software implementation on a GPP of a SAD routine, against the

execution of the same routine over the SAD arithmetic accelerator unit.

In Chapter 8, titled “**Conclusions**”, we conclude the dissertation summarizing our findings and we discuss the main contributions and suggestions for future research directions.

In Appendix A, we provide a brief overview of the most popular reconfigurable architectures. It provides background for the results presented in Chapter 7.

Chapter 2

Arithmetic Unit for collapsed SAD and Multiplication operations (AUSM)

Multimedia instruction set architectures (ISA) provide new demands on multi-operand addition related operations. Furthermore, when designing multimedia reconfigurable extensions [16, 41], special attention has to be paid to dynamic reconfiguration of arithmetic units. It is desirable that parts of hardware, common to several operations, can be configured in advance where the “differences”, rather than the entire unit, are adapted for reconfiguration. This motivates us to address the design problems of universal arithmetic units that perform multiple operations reusing common hardware blocks for different number representations. An Arithmetic Unit for collapsed SAD and Multiplication operations (AUSM) is proposed in this chapter. It reintroduces universal and collapsed units [42] and addresses additional problems imposed by multi-operand operations that require rectangular arrays, e.g. SAD implementations [43–46]. We consider an *arithmetic accelerator* supporting multiple number integer representations. Moreover, we assume unsigned, signed magnitude and two’s complement number notations into a single collapsed multiplier/SAD design.

This chapter is organized as follows. Section 2.1 presents the background for the proposed reconfigurable unit. Section 2.2 outlines the AUSM organization. Section 2.3 presents the experimental results of the mapped unit, as well as a comparison with other well know multiplier organizations in terms of used

area and delay. Finally, Section 2.4 ends with the conclusions.

2.1 Unit Collapsing Example

Single bit addition is typically implemented by a circuit called Full Adder (FA), also referred to as (3:2)counter in this thesis. Full adders are used as building blocks to implement a variety of more complex addition-based arithmetic functions. One basic structure for multi-bit addition is the Ripple Carry Adder (RCA) depicted in Figure 2.1. Considering the delay of one FA as the basic delay unit, an n -bit RCA will have n -FA delays or a latency of $\Theta(n)$ imposed by the carry propagation chain. The Carry Propagation Adder (CPA) latency problem has been addressed in several designs (see e.g. [47–49]). Nevertheless, the carry circuit on CPA, still imposes the main performance bottleneck in simple two operand addition operations.

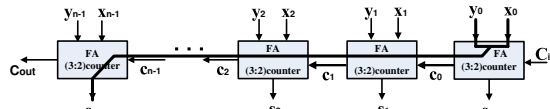


Figure 2.1: Ripple Carry Adder (RCA)

The above adder is not efficient for multiple operand additions. Applications that require addition of more than two operands can benefit from counter-based schemes used to reduce the total latency. The latency cost of adding n operands is reduced to $(n - 2)$ (3:2)counters delays plus one CPA delay for a multiplication operation. Figure 2.2 presents an example of a fixed point multiplication unit based on a (3:2)counter scheme (carry-save adder structure).

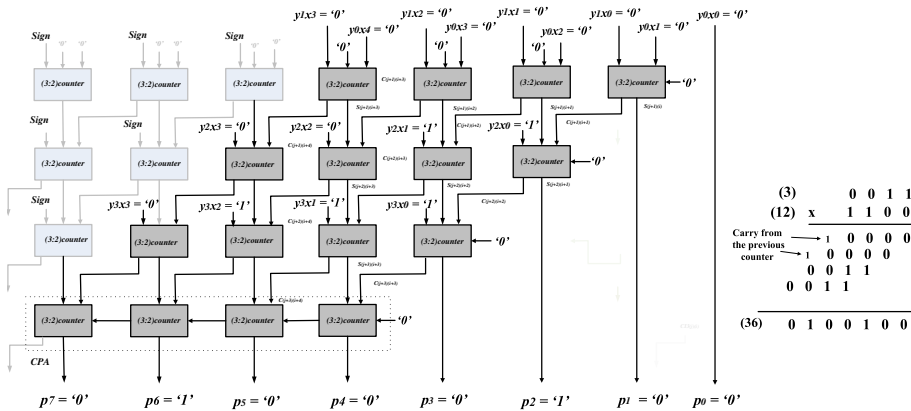


Figure 2.2: Basic array multiplier unit

The group of (3:2)counters presented in dark shaded rectangles in Figure 2.2 are used for unsigned numbers. When two's complement representation is considered, extra hardware is required for sign extension. These are the light shaded rectangles ((3:2)counters) in the leftmost part of Figure 2.2. An example of two's complement multiplication of the numbers -1 and -7 is presented in Figure 2.3. The additional (3:2)counters required to add the (*Sign*), necessary operation when consider sign extended multiplication schemes [50] are shown separately at the left.

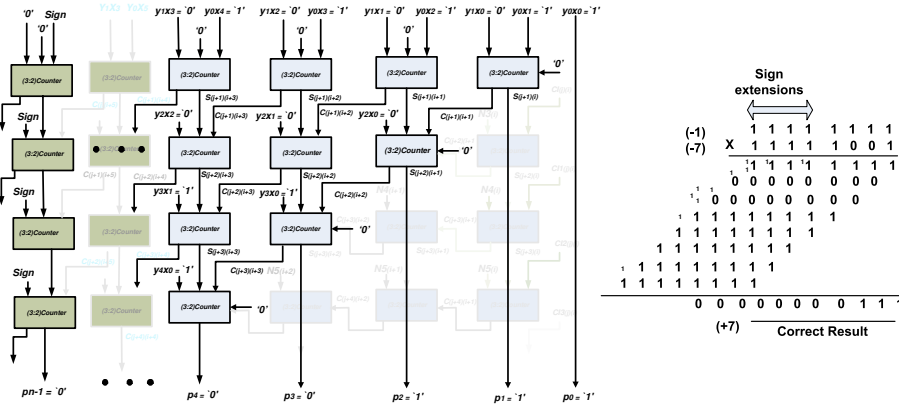


Figure 2.3: Two's complement sign extension scheme and example

Sum of absolute differences: The basic structure presented in the Figure 2.2 can be used to accelerate the processing of multiple-operand additions. An example of such operation, widely used in multimedia motion estimation, is the Sum of Absolute Differences (SAD). Motion estimation techniques divide the image frame in macro-blocks of size $n * n$ picture elements (pels). The algorithm establishes if there is a difference between two image blocks using the SAD operation. This is established by computing the absolute value differences between two picture elements of the *reference frame* and the *current-search* frame [46]. Commonly the sum of absolute differences is computed for the entire block performing the following operations: 1) find the largest, 2) find the smallest, 3) perform the subtraction, always subtracting a small value from the largest one (absolute difference) and 4) accumulate the results. The SAD operation is formally represented as follows:

$$SAD(x, y, r, s) = \sum_{j=1}^{16} \sum_{i=1}^{16} |A(x + i, y + j) - B((x + r) + i, (y + s) + j)| \quad (2.1)$$

where, the tuple (x, y) represents the position of the current block and (r, s)

denotes the displacement of B (pel of the search area), relative to the reference block A . Given that SAD's performance is critical, multimedia instruction sets have incorporated dedicated SAD instructions, and numerous schemes have been proposed to speed up the SAD operation [43, 51–53]. In this dissertation, we assume the scheme proposed in [46] because it separates the multi-operand addition from the determination of which operand should be subtracted to produce in parallel the sum of absolute values.

Notations: In this thesis the following notations are used to better define the operations of Chapters 2, 3 and 4. The notation in Chapter 5 is different to the definitions presented in this chapter, and similar to the sign-magnitude adder described in [54].

- $A[N]$ - an N -element array (matrix).
 - $a_{16}(j)$ - a 16-bit (row) element with position j in matrix A .
 - $a_{(j,i)}$ - bit i in the j -th (row) element of matrix A .
- In other words:

$$A[8] = A = \begin{bmatrix} a_{16}(7) \\ \cdot \\ \cdot \\ \cdot \\ a_{16}(0) \end{bmatrix} \equiv \begin{bmatrix} a_{(7,15)} & & & a_{(7,0)} \\ \cdot & \cdot & & \\ \cdot & & \cdot & \\ \cdot & & & \cdot \\ a_{(0,15)} & & & a_{(0,0)} \end{bmatrix}$$

Using the above notation, the SAD operation can be represented as:

$$SAD(A[N], B[N]) = \sum_{j=0}^{n-1} |a_{16}(j) - b_{16}(j)| \quad (2.2)$$

For the simplicity of the notations, hereafter we merge the two input matrices into one larger matrix in the following way:

$$IN_{[2N]} = A_{[N]} \cup B_{[N]} = \begin{bmatrix} a_{16}(N-1) \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ a_{16}(0) \end{bmatrix} \cup \begin{bmatrix} b_{16}(N-1) \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ b_{16}(0) \end{bmatrix} = \begin{bmatrix} a_{16}(N-1) \\ b_{16}(N-1) \\ \cdot \\ \cdot \\ a_{16}(0) \\ b_{16}(0) \end{bmatrix} = \begin{bmatrix} in_{16}(2N-1) \\ in_{16}(2N-2) \\ \cdot \\ \cdot \\ in_{16}(1) \\ in_{16}(0) \end{bmatrix}$$

SAD example: As an example, in Figure 2.4 (a) we depict the (3:2)counter array structure for adding six numbers. Three of these numbers are represented in one’s complement notation. Three Hot ones (‘1’) are added using the carry in the first column of counters. This is necessary to perform a subtraction in two’s complement representation. Figure 2.4 (b), presents the additional hardware coupled with the main (3:2)counter array. This entitled “carry unit” is used for a selective complementing (one’s complement) of the inputs. Figure 2.4(c) presents an example of the operations performed by the aforementioned hardware. In the example from Figure 2.4, we use decimal numbers and illustrate the functionality of our implementation:

$$|in_{16}(5) - in_{16}(4)| + |in_{16}(3) - in_{16}(2)| + |in_{16}(1) - in_{16}(0)| = |7 - 5| + |6 - 4| + |3 - 2| = 5$$

where: $IN[6] = A[3] \cup B[3]$,

$$A[3] = \begin{bmatrix} 7 \\ 6 \\ 3 \end{bmatrix}; B[3] = \begin{bmatrix} 5 \\ 4 \\ 2 \end{bmatrix} \Rightarrow IN[6] = [7, 5, 6, 4, 3, 2]^T$$

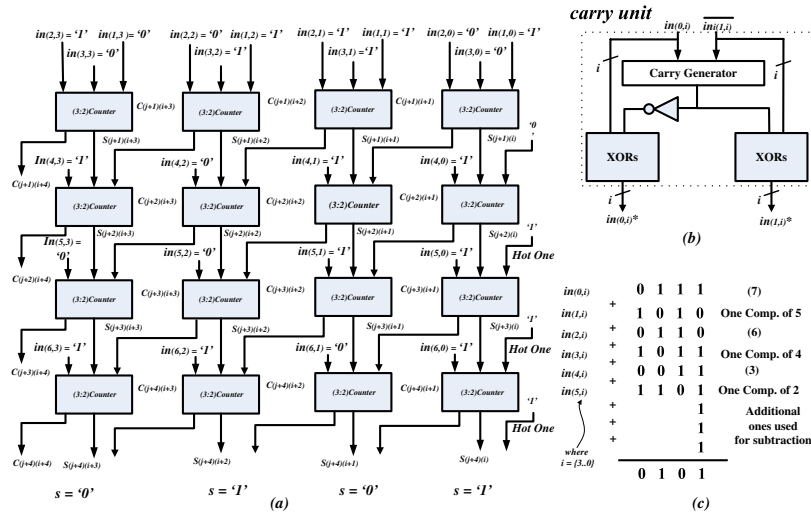


Figure 2.4: SAD (3:2)counter array

From Figures 2.2 and 2.4 one can conclude that portions of the (3:2)counter array can be reused to support two operations: the multiplication and the sum of absolute differences. Figure 2.5 presents an example of a group of (3:2)counters into the colored frame, commonly used to support both, two’s complement multiplication and SAD.

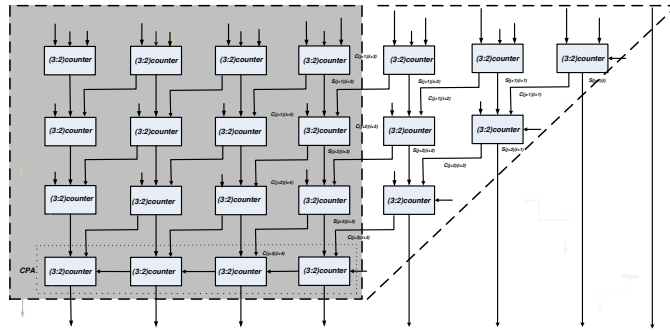


Figure 2.5: Common basic array

The example presented in Figure 2.5 illustrates our collapsing approach of several arithmetic operations (e.g. unsigned, signed magnitude and two's complement multiplication and SAD) into one universal arithmetic unit. A group of multiplexers enables the sharing of the (3:2)counter resources. The main purpose of these logic elements is to determine the operands to be issued into the array. Those elements are also used to enable or disable carry propagation. Figure 2.6 illustrates the scheme for a (3:2)counter with two multiplexers attached to enable the proposed functionalities. The multiplexer inputs a, b, \dots, n in $MUX A$ are e.g. SAD input and partial product inputs. $MUX B$ is used to force a C_{in} equal to zero or one; also it is used to propagate the (3:2)counter generated carry value

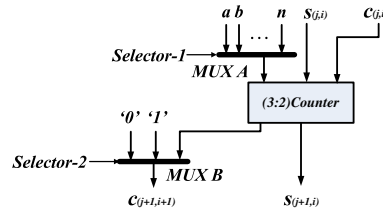


Figure 2.6: Cell detail in Figure 2.5

The selector signals are generated by a control unit and are used to set up the data path. *Selector-1* in $MUX A$ is used to choose one of the inputs (a, b, \dots, n), while the *Selector-2* in $MUX B$ forces a carry = '1' (e.g. Hot one) for two's complement notations. *Selector-2* is also used for disable the carry ('0') in case of SAD processing and to enable the propagation of a generated carry in multiplication operations (see details further).

2.2 The AUSM Array Organization

We present an adaptable arithmetic unit that collapses various multi-operand addition related operations into a single array. More precisely, we consider multiplication and SAD, and propose an array of processing elements capable of performing the aforementioned operations for unsigned, signed magnitude, and two's complement representations. The main idea is to compute several operations in parallel (SAD case), which increases substantially the performance compared to related works that perform those operations in a sequential processing way [55, 56]. In our case, we assume 16-bit integer numbers for both the multiplication and SAD operations. With appropriate considerations and without loss of generality, our approach can be extended to any desired data width. The main approach of collapsing operations into one unit was shown through examples in Section 2.1. As indicated earlier, we consider three number representations, unsigned, signed magnitude and two's complement. We note that when considering the array, each symbol in Figure 2.7 represents a (3:2)counter. It is actually an elaborated version of the example from Figure 2.5. Note that the common part is further divided into two triangular *sections* 2 and 3 on Figure 2.7.

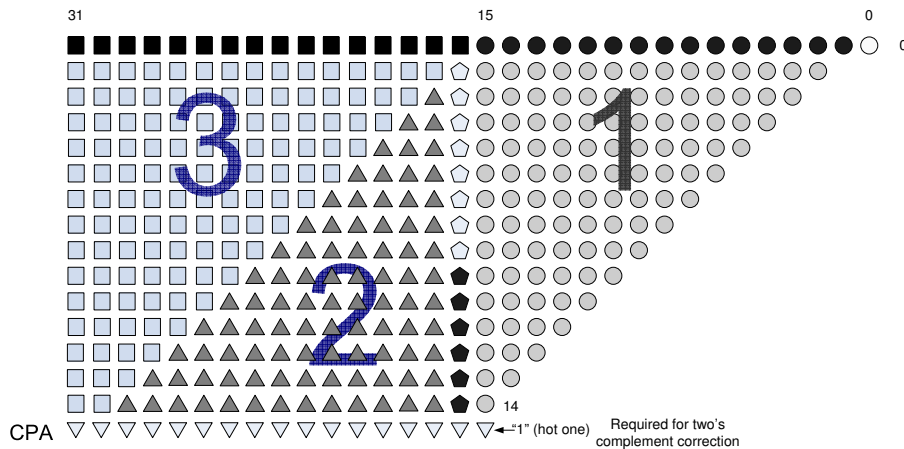


Figure 2.7: The AUSM: a 16x16 integer multiplication and SAD operation.

In Figure 2.7, the AUSM array columns are enumerated from 0 to 31, it has 0 to 14 rows, and the array is divided into three main *sections* detailed further. The last row represent the carry propagating adder at the end of the

(3:2)counter tree. Let the multiplier and the multiplicand be denoted by $y_n(0)$ and $x_n(0)$:

$$y_n(0) = y_{(0,n-1)}y_{(0,n-2)}\cdots y_{(0,2)}y_{(0,1)}y_{(0,0)}$$

$$x_n(0) = x_{(0,n-1)}x_{(0,n-2)}\cdots x_{(0,2)}x_{(0,1)}x_{(0,0)}$$

Our 16-bit universal multiplier is constructed around a unit composed by the following components:

a) Partial product (pp)¹ generation:

$$pp_{(j,i)} = x_{(0,i)} \cdot y_{(0,j)},$$

$$\forall : [x_{16}(0) = x_{(0,15)}\cdots x_{(0,0)}] \text{ and } [y_{16}(0) = y_{(0,15)}\cdots y_{(0,0)}].$$

b) The partial product addition, based on a (3:2)counter organization, and

c) Final adder with carry propagation.

The first row in Figure 2.7, denoted by the black circles, receives the first group of partial products of the multiplier ($x_{(0,0)}y_{(0,i)} \quad \forall 1 \leq i < 15$) and ($x_{(0,1)}y_{(0,i)} \quad \forall 0 \leq i < 14$). The other 14 partial products are accommodated in the remaining 14 rows of the array corresponding to *sections 1* and *2*. *Section 3* is used for sign extension in multiplication operations. In conjunction with *section 2*, *section 3* is also used by the SAD processing.

Multiplication in universal notations: The accommodation of different multiplication operation types into the array is accomplished with changes in the sign values for signed magnitude representations, and with sign extension for two's complement representations. In the sign-magnitude notation we denote the Most Significant Bits (MSBs), which are the sign bits, as $x_{(0,n-1)}$ and $y_{(0,n-1)}$. Table 2.1 presents in column 4 the three types of extensions needed to properly perform multiplication for all unsigned, signed-magnitude and two's complement notations. Unsigned numbers and signed-magnitude number are extended with zero, while two's complement are extended with the sign bit extension along the (3:2)counters of *section 3* (see Figure 2.7). Also, Table 2.1 indicates that in unsigned and two's complement notations, no changes are introduced in the Most Significant Bits (MSBs) for computing

¹An "AND" gate is used for two bit multiplication. The multiplication of two n-bit numbers e.g. x and y , will require n^2 2-input "AND" gates. The partial products generated by the "AND" are then added

purposes as shown in columns 2 and 3. On the other hand, in signed-magnitude numbers, MSB is forced to be zero (sign values = 0). At the end of the operation, the final multiplication sign is corrected by replacing the computed value with the result of the XOR operation between $x_{(0,n-1)}$ and $y_{(0,n-1)}$ (e.g., $x_{(0,n-1)} \oplus y_{(0,n-1)}$) as shown in column 5 of Table 2.1.

Table 2.1: Universal multiplier extensions - MSB use in addition operation

	$x_{(0,n-1)}$ (MSB of x_n)	$y_{(0,n-1)}$ (MSB of y_n)	Extension (section 3)	Sign correction
Unsigned	$x_{(0,n-1)}$	$y_{(0,n-1)}$	0	N.A.
Signed Magnitude	0	0	0	$x_{(0,n-1)} \oplus y_{(0,n-1)}$ (updated value)
Two's complement	$x_{(0,n-1)}$	$y_{(0,n-1)}$	Sign extension: $y_{16}(0) \cdot x_{(0,n-1)}$	N.A.

Sum of absolute difference processing: The SAD calculation can be decomposed into two logical steps. In order to perform a universal operation, we need to produce a positive result². To achieve such a goal, we proceed as follows:

1. We note that SAD inputs are positive integers. *The first step determines which of the operands is greater, so that the smallest is subtracted in the array to produce the absolute values at once.* The logic required to perform this kind of functionality (comparison) is obtained from the carry out (Cout) of the addition of one operand $in_{(0,i)}$ and the inverted operand $in_{(1,i)}$ (see Figure 2.4(b)), where the subindex '0' or '1' represents input 0 and input 1 respectively and i indicates that the binary number has i bits. Thus, the carry out indicates if $in_{(0,i)}$ or $in_{(1,i)}$ is the greatest and is computed into the entitled "carry unit". This is true because of the following:
 - a:** The sign bit for sign magnitude and two's complement is 0, (inputs to SAD are positive) thus they are equal.
 - b:** For all the bits for the unsigned numbers and the magnitude of the signed magnitude and two's complement numbers; for the first

²Positive numbers and unsigned notation numbers have the same representation and can be seen as positive numbers with an implicit 0 sign-bit.

operand $in_{(0,i)}$ to be greater, then the second operand $in_{(1,i)}$ must be that all most significant bits are equal (sign included, see a; this is to make equal length operands and to have the same carry circuitry for all notations) and that there is a bit position x such that the bit value of $in_{(0,i)}$ is 1 and $in_{(1,i)}$ is 0. The reason is the following: if the bit position is x , then starting at position x and ending at position 0, $in_{(0,i)} = 2^x$ is the worst case (the rest of the bit starting at $x-1$ are all 0) and $in_{(1,i)} = 2^x - 1$ is the best case (all remaining least significant bits are 1). Consequently, when inverting $in_{(1,i)}$ all most significant bits starting from $x + 1$ have opposite values for both $in_{(0,i)}$ and $in_{(1,i)}$ and at the bit position x , $in_{(0,x)} = 1$ and $in_{(1,x)} = 1$, implying that at position x a carry will be generated and it will be transmitted out, consequently $Carry-Out = 1$. If $in_{(1,i)}$ is greater, then $in_{(0,i)} = 0$ and $in_{(1,i)} = 1$. Thus $\overline{in_{(1,i)}} = 0$ and a potential carry from the least significant bit is killed. Also because the most significant bits starting at position $x + 1$ have opposite values there is no generated carry thus $Carry-Out = 0$. If $in_{(0,i)}$ and $in_{(1,i)}$ are equal then the carry out is zero.

In summary, the absolute operation $|in_{(0,i)} - in_{(1,i)}|$ can be substituted with $in_{(0,i)} - in_{(1,i)}$ or $in_{(1,i)} - in_{(0,i)}$, depending whether $in_{(0,i)}$ or $in_{(1,i)}$ is the smallest and thus obtaining a positive result. For this, one of the operands is one's complemented and then the carry out of the addition of both operands is computed, as stated mathematically by the following equations:

$$\overline{in_{(0,i)}} + in_{(1,i)} \geq 2^i - 1 \quad (2.3)$$

therefore

$$in_{(1,i)} > in_{(0,i)} \quad (2.4)$$

means checking whether the addition of the binary complement of $in_{(0,i)}$ and the operand $in_{(1,i)}$ produces a carry out. The outcome determines which is the smallest, depending on the existence or not of the carry output as described in the example presented in Figure 2.4(b) with the "carry unit".

2. The second step creates the array for multi-operand additions using *section 2* and *3*, corresponding to the (3:2)counters enumerated 16 to 31

inclusive (see Figure 2.7). Sixteen operands are received in this array, eight corresponding to the reference block and the other 8 come from the block of the search area (SAD processing). Those operands are pre-processed into a set of 8 carry units. For example $in_{(0,i)}$ and $in_{(1,i)}$ enter the multi-operand addition tree of (3:2)counters after being complemented selectively into a carry unit. In the same way, the other 14 inputs are preprocessed in pairs; those inputs are also selectively complemented into the carry units before entering the multiple-operand addition tree. Therefore, the square array of (3:2)counters constructed with sections 2 and 3, receive 16 inputs of 16 bits width each one, and process concurrently the half-block of a 4×4 SAD operations. The following matrix IN is used to represent the 16 inputs elements of 16-bits each for SAD processing.

$$IN[16] = \begin{bmatrix} in_{16}(15) \\ \cdot \\ \cdot \\ \cdot \\ in_{16}(0) \end{bmatrix} \equiv \begin{bmatrix} in_{(15,15)} & \cdot & \cdot & \cdot & in_{(15,0)} \\ & \cdot & & & \\ & & \cdot & & \\ & & & \cdot & \\ in_{(0,15)} & \cdot & \cdot & \cdot & in_{(0,0)} \end{bmatrix}$$

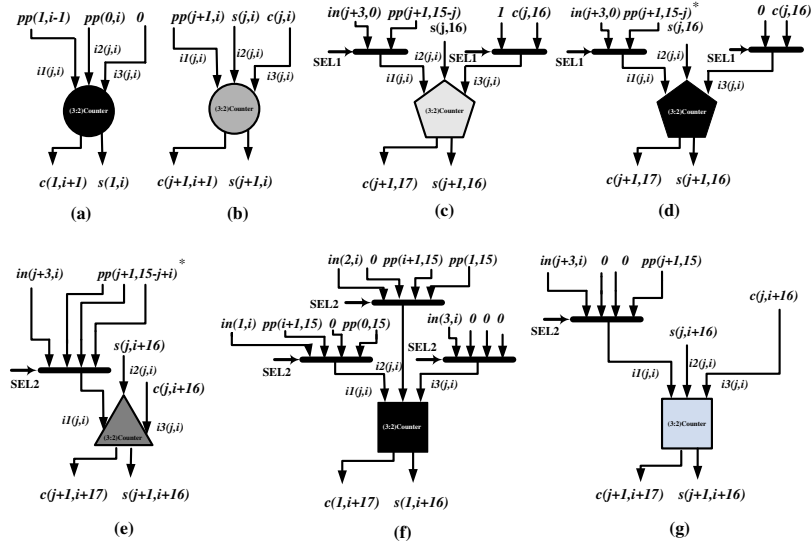
The array description: As indicated earlier, each block in Figure 2.7 is a (3:2)counter. Figure 2.8 details the different kinds of logical blocks used in our implementation. Some of those blocks utilize multiplexers denoted in the figure as thick horizontal bars. The control signals SEL1 and SEL2 drive the multiplexers to feed the (3:2)counters with correct data. SEL1 is a one-bit and SEL2 is a two-bit multiplexer selection signal. A decoder, attached to any instruction for this unit, should consider the Table 2.2.

Table 2.2: Selector signal SEL1 and SEL2 - Multiplexer behavior

Selector signal	Function
SEL1 $\equiv 0_2$	It is used to select the SAD operands.
SEL1 $\equiv 1_2$	Route partial products for multiplication operations.
SEL2 $\equiv 00_2$	It is used to select the SAD operands.
SEL2 $\equiv 01_2$	Route the data for unsigned multiplication operation.
SEL2 $\equiv 10_2$	Used to choose the signed-magnitude multiplication data.
SEL2 $\equiv 01_2$	Used to select the two's complement values.

It has to be noticed that the empty circle presented in the first row of Figure 2.7 (first symbol in the upper right corner), represents the product $x_{(0,0)} \cdot y_{(0,0)}$ that

it is computed by an 2-input AND gate and does not contain any counter.



* : PC is complemented in the last row; and Cin = 1 in the CPA for a negative two's complement multiplier.

Figure 2.8: (3:2)counters of the AUSM array

Table 2.3 presents in the second column the operation supported by the different type of logic blocks depicted in Figure 2.8. The third column presents the number of (3:2)counters used in each block type.

Table 2.3: (3:2)counters used in the AUSM scheme

Counter type see Figure 2.8	Operation	Number of counters used
(a)	Multiplication	15
(b)	Multiplication	105
(c)	Multiplication and SAD	8
(d)	Multiplication and SAD	6
(e)	Multiplication and SAD	91
(f)	Multiplication and SAD	16
(g)	Multiplication and SAD	119

Finally, we describe the inputs for the blocks depicted in Figure 2.8. We consider that the (3:2)counters have the followings three inputs $i1_{(j,i)}$, $i2_{(j,i)}$ and

$i3_{(j,i)}$ and produce two outputs $s_{(j,i)}$ that correspond to the SUM, and $c_{(j,i)}$ for the CARRY output; where those outputs are a function of their corresponding inputs ($[c_{(j,i)}, s_{(j,i)}] = f[i1_{(j,i)}, i2_{(j,i)}, i3_{(j,i)}]$). A concise description of the index ranges of the logic blocks elements presented in Figure 2.8, is shown through the following pseudo-codes ³:

```

(a): for  $i = 1..15$ 
         $i1_{(0,i)} = pp_{(0,i)}$ ,
         $i2_{(0,i)} = pp_{(1,i-1)}$ ,
         $i3_{(0,i)} = 0$ 
    end

(b): for  $j = 1..14$ 
         $n = j + 1$ ,
        for  $i = n..15$ 
             $i1_{(j,i)} = pp_{(j+1,i)}$ ,
             $i2_{(j,i)} = s_{(j,i)}$ ,
             $i3_{(j,i)} = c_{(j,i)}$ ;
        end
    end

(c): for  $j = 1..8$ 
         $i1_{(j,16)} = in_{(j+2,0)} \cdot (SEL1 \equiv 0) + pp_{(j+1,15-j)} \cdot (SEL1 \equiv 1)$ ,
         $i2_{(j,16)} = s_{(j,16)}$ ,
         $i3_{(j,16)} = 1 \cdot (SEL1 \equiv 0) + c_{(j,16)} \cdot (SEL1 \equiv 1)$ 
    end

(d): for  $j = 9..14$ 
         $i1_{(j,16)} = in_{(j+2,0)} \cdot (SEL1 \equiv 0) + pp_{(j+1,15-j)} \cdot (SEL1 \equiv 1)$ ,
         $i2_{(j,16)} = s_{(j,16)}$ ,
         $i3_{(j,16)} = 0 \cdot (SEL0 \equiv 0) + c_{(j,16)} \cdot (SEL0 \equiv 1)$ 
    end

(e): for  $j = 2..14$ 
         $n = j - 1$ ;
        for  $i = 1..n$ 
             $i1_{(j,i+16)} = in_{(j+2,0)} \cdot (SEL2 \equiv "00") + pp_{(j+1,15-j+i)} \cdot (SEL2 \equiv "01") +$ 
                 $pp_{(j+1,15-j+i)} \cdot (SEL2 \equiv "10") +$ 
                 $pp_{(j+1,15-j+i)} \cdot (SEL2 \equiv "11")$ 

             $i2_{(j,i+16)} = s_{(j,i+16)}$ ,
             $i3_{(j,i+16)} = 1 \cdot (SEL2 \equiv "00") + c_{(j,i+16)} \cdot (SEL2 \equiv "01")$ 
        end
    end

```

³by $SEL1 \equiv 0$, we denote an expression which is true when signal SEL1 has value 0₂, e.g., ($SEL1 \equiv 0_2 = \text{TRUE}$)

(f): for $i = 0..15$

$$i1_{(0,i+16)} = in_{(0,i)} \cdot (SEL2 \equiv "00") + pp_{(i+1,15)} \cdot (SEL2 \equiv "01") + pp_{(0,15)} \cdot (SEL2 \equiv "11")$$

$$i2_{(0,i+16)} = in_{(1,i)} \cdot (SEL2 \equiv "00") + pp_{(i+1,15)} \cdot (SEL2 \equiv "10") + pp_{(1,15)} \cdot (SEL2 \equiv "11")$$

$$i3_{(0,i+16)} = in_{(2,i)}$$

end

(g): for $j = 1..14$

for $i = j..15$

$$i1_{(j,i+16)} = in_{(j+2,i)} \cdot (SEL2 \equiv "00") + pp_{(j+1,15)} \cdot (SEL2 \equiv "11"),$$

$$i2_{(j,i+16)} = s_{(j,i+16)},$$

$$i3_{(j,i+16)} = c_{(j,i+16)};$$

end

end

The following must also be considered to ensure correct results when two's complement representation is used: $pp_{(15,i)} = pp_{(15,i)} \text{ XOR } pp_{(15,15)}$ and $Cin = 0 \text{ XOR } pp_{(15,15)}$; this is necessary to produce "hot one" addition effect needed for two's complement correction.

2.3 Experimental Results

The AUSM including the carry unit were implemented using VHDL, synthesized, functionally tested, and validated using the ISE 5.2 Xilinx environment [57] and Modelsim [58], for the VIRTEX II PRO FPGA device. The unit's features include the following:

- One 16 x 16 multiplier for operate with unsigned, signed-magnitude and two's complement representation as well as for receive 8-input pairs of 16-bits SAD operands in the same AUSM array;
- A latency of 23.9 ns for processing a 16 bit multiplication and 29.88 ns for processing 8-input pairs of SAD operands, for unsigned, signed magnitude and two's complement representations;
- It is estimated that 2/3 of the resources are shared by the multiplication and SAD functionalities in universal representation. This sharing could be of benefit when dynamically reconfiguring environments are considered, based on frames differences.

Furthermore, a classic unsigned array multiplier and a Baugh and Wooley (B&W) signed two's complement multiplier [59] were implemented and synthesized using the same tools for comparison reasons. For all implementations, we use a ripple carry adder in the final stage. We have also implemented and synthesized parallel additions and consider the fast carry support of the Xilinx Virtex II PRO technology. Table 2.4 summarizes the performance of these structures.

Table 2.4: AUSM and other multiplication units.

- Latency -

Unit	Logic	Wire	Total
Unsigned M. [50] ‡	14.589 ns 49.8%	14.639 ns 50.2%	29.282 ns 100%
Baugh Wooley [59] ‡	15.555 ns 49.6%	15.826 ns 50.4%	31.381 ns 100%
our proposal ‡ (LUT based)	15.877 ns 50.2%	15.741 ns 49.8%	31.618 ns 100%
our proposal § with CLA	16.112 ns 54.2%	13.603 ns 45.8%	29,715 ns 100%
our proposal (RCA-Xilinx based unit)	14.311 ns 59.9%	9.568 ns 40.1%	23.879 ns 100%
Carry Unit (into the RCA-Xilinx based unit)	2.576 ns 43.6 %	3.338 ns 56.4 %	5.914 ns 100 %

‡ : RCA as a final adder; LUT based implementation

§ CLA: Carry Lookahead Adder as final adder; LUT implementation

It can be noticed that the proposed array incorporates additional logic and it is expected to perform somehow slower than the other multiplier units. It is observed that both, our proposal and Baugh&Wooley's proposal for two's complement numbers, are a bit slower than the classic unsigned multiplier. There are negligible differences in timing between our AUSM proposal and the Baugh&Wooley's two's complement multiplier.

The time delay introduced by the multiplexers used to feed the (3:2)counters in order to perform the four operations has a constant delay for all blocks of the array; therefore, from the obtained results it is evident that the routing delay is still significant in reconfigurable devices. Additionally, regarding the carry unit required for SAD, it is observed that 5.914 ns are needed for the

processing of the carry-out and the inversion through the XOR gates. The latency presented in Table 2.4 for the carry unit, corresponds to the latency of our proposal constructed with the support of the RCA with the use of the hard IPs of Virtex II PRO devices (RCA-Xilinx) [60].

As expected, concerning the silicon used by the proposed unit when compared with the other structures, the added functionalities require some extra resources as depicted on Table 2.5.

Table 2.5: AUSM and other multiplication units
- Hardware Use -

Unit	# Slices	# LUTs	# IOBs
Unsigned M. [50] ‡	300	524	64
Baugh & Wooley [59] ‡	330	574	65
our proposal ‡ (LUT based)	686	1198	322
our proposal § with CLA	711	1244	322
our proposal RCA-Xilinx based unit	658	1170	322
Carry Unit (into the RCA-Xilinx based unit)	8	16	32

‡: RCA as a final adder; LUT based implementation

§ CLA: Carry Lookahead Adder as final adder; LUT implementation

The carry unit used in SAD operations consumes a considerable hardware for routing the correct operands into the universal array suggested by Table 2.5. The 16 data entries consume 128 slices, 8 slices per unit. This amount of hardware represents an 18 % of the slices used in the proposed unit. In spite of that, this constitutes only 1 % of the VIRTEX II PRO resources device.

When reconfigurable platforms with partially reconfiguration support are considered, it is of interest to set only the differences rather than configure at once the entire structure. For example, pre-configure the basic array of (3:2)counters and then modify the necessary elements to transform the array into a multiplier or a SAD unit, e.g., introduce XOR gates for complement conditionally the data or introduce some bypass multiplexors. Our calculations indicate that 66.6 % of the basic array are shared by the supported operations. Indeed we

have a multiplexer, which changes the functionality of the unit into one of the collapsed operations. This makes the unit reprogrammable and suitable for ASIC implementation as the minimum hardware complexity for a set of operations. Furthermore, considering contemporary reconfigurable technologies, with partial configuration capabilities, our proposal can be more efficient both in hardware utilization and in performance. This can be achieved as follows: 1) initially configure the FPGA only with the common functionality of the collapsed unit and 2) later during run time, partially reconfigure, to adapt the basic unit towards a particular operation, minimizing with this the total configuration time and FPGA utilization. We estimated using partial reconfiguration [61] that the reconfiguration frames difference between the functionalities proposed could be around 50 %, improving in this way the latency required over complete reconfigurations.

2.4 Conclusions

In this chapter, a detailed description of our universal unit for the processing of the sum of absolute differences and a multiplier operating with unsigned, signed magnitude and two's complement representation has been presented. A brief analysis of the time cost associated to the implementation indicates that the proposed approach regardless of the additional function reveals a similar performance in terms of time delay when compared with a single two's complement multiplication schemes.

The proposed unit is capable to process an 8-pair-input SAD operation in 29.8 ns; and takes 23.9 ns for a 16-bit multiplication when targeting Virtex II Pro-6 FPGA technology. When our AUSM scheme is compared to multiplication units for two's complement notation (Baugh & Wooley), we observed that it incurs in negligible extra delays, but performing additional functionalities such as: SAD and multiplication for unsigned, two's complement and sign magnitude integer numbers. Our experimental results suggest that instead of reconfiguring the complete unit into the FPGA device (e.g. a multiplier or a SAD), we only need 50 % of the configuration stream data of the original one, when we want to migrate from an instantiated multiplier to the SAD unit (partial reconfiguration). In this way, the set up time for adaptation between different units such as a multiplication unit and an 8-pair-input SAD unit can be reduced.

Chapter 3

AUSM extension

An arithmetic unit that collapses multiplication and Sum of Absolute Differences (SAD) was introduced in the previous chapter. In this chapter this unit is extended to support additional functionality. The new structure has rectangular shape and contains one additional (3:2)counter row right above the CPA (the last row) of the original AUSM structure. The rectangular array is used for collapsing the eight multi-operand addition related operations. For this unit, we consider multiplication in integer and fractional notations; the sum of absolute differences in unsigned, signed magnitude and two's complement notations. Furthermore, our proposal incorporates additional functionality in the Multiply-Accumulation (MAC) mode for two's complement representation. The extended AUSM scheme proposed in this chapter can be implemented in an ASIC as a run time parametrizable unit, or synthesized and mapped on reconfigurable technology. It is estimated that 6/8 of the basic (3:2)counter arrays in our design are re-used by the different operations supported. The obtained results on the presented unit indicate that it is capable of processing a 4×4 SAD macro-block in 36.35 ns and it takes 30.43 ns to process the remaining of the operations using a VIRTEX II PRO xc2vp100-7 FPGA device.

This chapter is organized as follows. Section 3.1 describes the proposed AUSM extension. Section 3.2 presents the mathematical equations supporting the proposed functionalities. Section 3.3 outlines the experimental results on the mapped unit, as well as other comparison units in terms of area utilization and delay. Finally, the chapter is concluded with Section 3.4.

3.1 The AUSM Extension: a Multiple-Addition Array

This section begins by presenting a brief background of the SAD operation that was widely discussed in the previous chapter. Further, an explanation of the AUSM extension is presented. Finally, a complete description of the equations' set for the construction of the proposed arithmetic accelerator unit is given.

3.1.1 Previous Work

Equation (3.1) is used for the motion estimation between two blocks.

$$SAD(x, y, r, s) = \sum_{j=1}^{16} \sum_{i=1}^{16} |A(x+i, y+j) - B((x+r)+i, (y+s)+j)| \quad (3.1)$$

where, the duple (x, y) represents the position of the current block, and the pair (r, s) denotes the displacement of B , relative to reference block A . Several methods have been proposed for speeding up the SAD kernel, e.g. the work presented in [44–46]. Those proposals use several Processing Elements (PE) such as: a set of subtractors and accumulation circuits to perform the SAD operation. The data is distributed over subtractor units and the result of this operation is issued into a regular accumulator circuits (with local feedback). Thus, each accumulator is in charge of iteratively adding the subtraction results of each pair of SAD terms. On the other hand, the proposal presented in this chapter follows the approach used in Chapter 2: we process several elements in parallel. The processing requires that SAD input terms received by the multiple operation array, in this case the extended AUSM, have to be ordered and complemented selectively before computation. The rectangular shape of the AUSM extension schemes allows the processing of 16 input pairs of SAD, a 4×4 macroblock, doubling the processing capacity achieved in the previous chapter. Additionally, the new organization allows the creation of a dedicated multiplication unit for fractional representation. Furthermore, an extra row (row 15) of (3:2)counters is augmented to the AUSM array. This extra row of counters enables an extra operand addition, functionality necessary for the setup of the multiplication and accumulation operation. This approach follows previous schemes such as the work presented in [62].

3.1.2 Array Sectioning

The extended AUSM unit is constructed around a rectangular array of (3:2)counters. The general idea is illustrated in Figure 3.1. A set of (3:2)counters presented in grey in Figure 3.1(b) is added to the basic AUSM array depicted in Figure 3.1(a), augmenting its processing capabilities while preserving the original AUSM functionality.

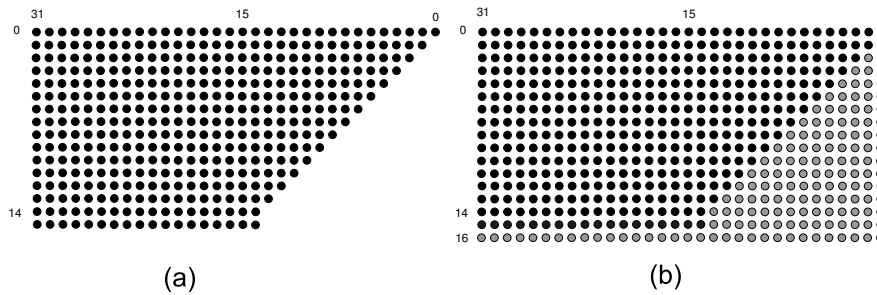


Figure 3.1: Multiple operation units: a) AUSM array b) AUSM extended array

The proposed new rectangular array consist of ten operational *sections* presented in Figure 3.2.

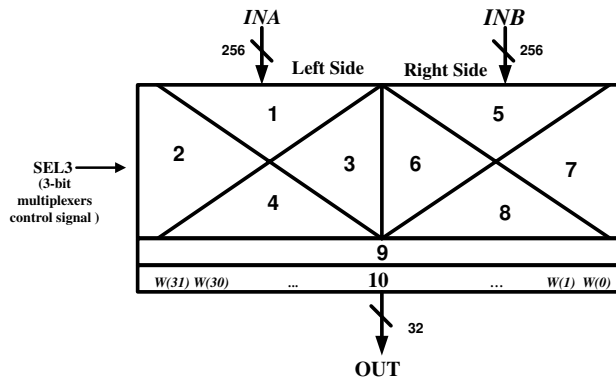


Figure 3.2: The AUSM extension array scheme

Depending on the particular operation, the inputs INA and INB accommodate input operands described in Table 3.1.

Table 3.1: Input and output of the AUSM extension array

Operation	Operand <i>INA</i>	Operand <i>INB</i>	Result OUT
Unsigned integer multiplication	$x_{16}(0)$ (16-bit integer)	$y_{16}(0)$ (16-bit integer)	$p_{32}(0)$ (32-bit integer)
Sign-magnitude integer multiplication	$x_{16}(0)$ (16-bit integer)	$y_{16}(0)$ (16-bit integer)	$p_{32}(0)$ (32-bit integer)
Two's complement integer multiplication	$x_{16}(0)$ (16-bit integer)	$y_{16}(0)$ (16-bit integer)	$p_{32}(0)$ (32-bit integer)
Two's complement integer MAC	$x_{16}(0)$ (16-bit integer) $w_{16}(0)$ (16-bit integer)	$y_{16}(0)$ 16-bit integer $w_{16}(0)$ (16-bit integer)	$mac_{32}(0)$ (32-bit integer)
Unsigned fractional multiplication	$a_{16}(0)$ (16-bit fractional)	$b_{16}(0)$ (16-bit fractional)	$p_{32}(0)$ (32-bit fractional)
Sign-magnitude fractional multiplication	$a_{16}(0)$ (16-bit fractional)	$b_{16}(0)$ (16-bit fractional)	$p_{32}(0)$ (32-bit fractional)
Two's complement fractional multiplication	$a_{16}(0)$ (16-bit fractional)	$b_{16}(0)$ (16-bit fractional)	$p_{32}(0)$ (32-bit fractional)
SAD	$INA[16]$ two interleaved input 2×4 blocks of 16-bit integers	$INB[16]$ two interleaved input 2×4 blocks of 16-bit integers	$\sum_{i=1}^8 ina_{16}(i) - ina_{16}(i+1) $ $\sum_{i=1}^8 inb_{16}(i) - inb_{16}(i+1) $ two 16-bit integers

Sections use

The 10 different *sections* of our extended AUSM array are used to set up the following functionalities:

- For integer multiplication, *sections* 5, 6, 3 and 4 are used to add the partial products. Additionally, *section* 9 is used to add the last partial products of the operation. Two's complement multiplications are processed with sign extension in *sections* 1 and 2. The same *sections* are used for the introduction of zero values, necessary to achieve the universal multiplication characteristics. Approach that is similar to the one used into the AUSM scheme presented in Chapter 2.
- Fractional multiplication operations are achieved through *sections* 1, 3, 6 and 8. Additionally, *sections* 2 and 4 are employed for adding the sign extensions for two's complement fractional representations. *Section* 2 and 4 are also used to add zeros for sign-magnitude processing.
- SAD processing is carried out into the left and right side of the AUSM extended array (see Figure 3.2). *Sections* 1, 2, 3 and 4 are used to set up the functionality proposed in the AUSM scheme. *Sections* 5, 6, 7 and 8 process one additional group of 8-SAD inputs. Thus, the computing capability is doubled. The 32 SAD terms required by the unit are represented through the matrix $IN[32]$:

$$IN[32] = \begin{bmatrix} in_{16}(32) \\ \cdot \\ \cdot \\ \cdot \\ in_{16}(1) \end{bmatrix} \equiv \begin{bmatrix} in_{(32,15)} & \cdot & \cdot & \cdot & in_{(32,0)} \\ & \cdot & & & \\ & & \cdot & & \\ & & & \cdot & \\ in_{(1,15)} & \cdot & \cdot & \cdot & in_{(1,0)} \end{bmatrix}$$

- Finally, an extra row of (3:2)counters is used for the addition of the 32-bit $w_{32}(0)$ operand. *Section* 10 provides the capability for processing MAC operations.

3.1.3 Array Organization

As mentioned before, the integer multiplication with universal characteristics is carried out in the AUSM extended scheme in a similar way as in the AUSM proposal presented in Chapter 2. The rectangular organization of the extended AUSM scheme allows the collapsing of two SAD units (similar to the one described in Section 2.1). Each SAD unit operates over a rectangular shape,

performing the operations described in Figure 2.4 (see Chapter 2). In the left side of the AUSM extension scheme depicted in Figure 3.2 the first SAD unit is instantiated. The right side is used to build the additional SAD unit. The multiplication of two fractional numbers resembles the multiplication of two integer numbers. Figure 3.3(a) and (b) illustrates an example of multiplication of two unsigned fractional numbers (9/16) and (5/16) with the (3:2)counter array and the operation performed by the aforementioned hardware. The proposed organization depicted in Figure 3.3 is an example of the large scale array instantiated into the AUSM extended scheme. The middle diagonal shape denoted with the highlighted (3:2)counters in Figure 3.3 performs fractional multiplication in our design. This unit is then collapsed into the rectangular array presented in Figure 3.2; specifically, the fractional multiplication array is composed of *sections 1, 2, 3, 4, 6 and 8*.

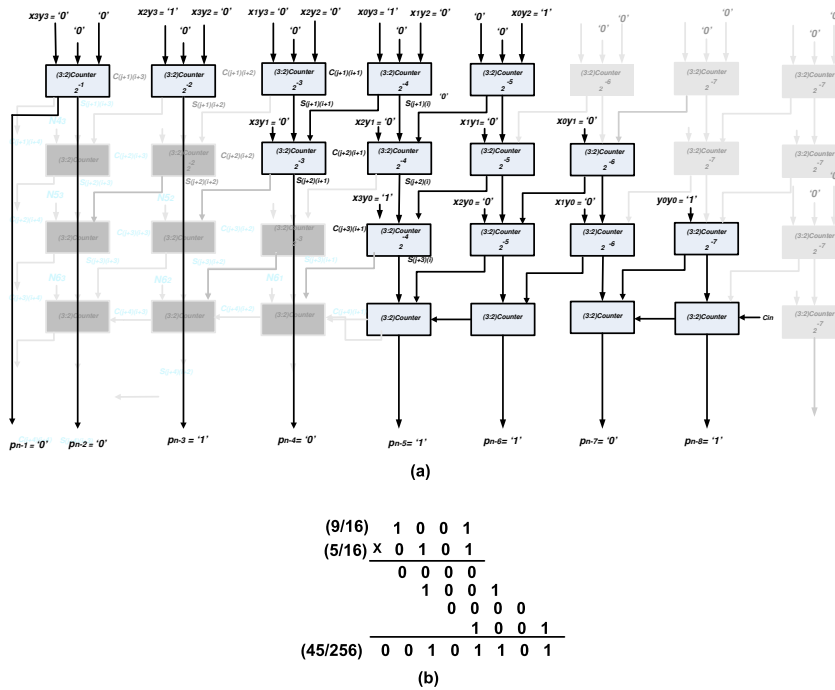


Figure 3.3: Fractional multiplication example.

The AUSM extension receives the integer multiplicand $x_{16}(0)$ and the integer multiplier $y_{16}(0)$, both represented in 16-bits. The augmented AUSM scheme receives also the fractional-numbers¹ $a_{16}(0)$ and $b_{16}(0)$; where the fractional

¹in terms of binary numbers, each magnitude bit represents a power of two, while each fractional bit represents an inverse power of two. e.g. $\frac{5}{16} = 0.0101 = 0x2^{-1} + 1x2^{-2} + 0x2^{-3} + 1x2^{-4}$

multiplicand $a_{16}(0)$ and the fractional multiplier $b_{16}(0)$ (also in 16 bits). A fractional fixed point binary number is defined as a number in the interval $-1,1$ expressed in a fixed number of bits. An N -bit fractional number can be expressed sign-magnitude (FSM) and two's complement (FCT) notations as follows:

$$FSM_N : b_{SM} = (-1)^{bo} \sum_{j=1}^{N-1} b_j 2^{-j} \text{ (signed magnitude)}$$

$$FCT_N : b_{TC} = -b_0 + \sum_{j=1}^{N-1} b_j 2^{-j} \text{ (two's complement)}$$

The processing of both, integer and fractional numbers require the partial product generation. AUSM extended scheme use $z_{(j,i)}$ and $f_{(j,i)}$ to represent the partial products of integer and fractional numbers respectively as stated by the following pseudo-code:

```

for  $0 \leq j \leq 15$ 
  for  $0 \leq i \leq 15$ 
     $z_{(j,i)} = x_{(0,i)} \cdot y_{(0,j)}$ 
     $f_{(j,i)} = a_{(0,i)} \cdot b_{(0,j)}$ 
  end
end

```

The (3:2)counter organization as presented in Figure 2.6 in Chapter 2, is also used in the extended AUSM scheme. Figure 3.4 presents the basic organization of the (3:2)counter pattern replicated in all sections of the array. The figure presents a group of (3:2)counters with inputs $i1_{(j,i)}$, $i2_{(j,i)}$ and $i3_{(j,i)}$ and two outputs $s_{(j,i)}$ and $c_{(j,i)}$. In this array, the input $i1_{(j,i)}$ receives the data through a multiplexer used to select adequate data depending on the operation performed, e.g., receive values such as: $in_{(j,i)}$ (addendum terms of SAD operation), or partial products terms such as $f_{(j,i)}$ and $z_{(j,i)}$. The inputs $i2_{(j,i)}$ and $i3_{(j,i)}$ receive the previous computed values of SUM and CARRY. In Figure 3.4, the multiplexers are presented as thick bars. Signal $SEL3$, which is detailed in the following section, is used for controlling the multiplexer, and for choosing the appropriate data for the proposed operations. $SEL3$ configures in this way the operation to be computed by the array.

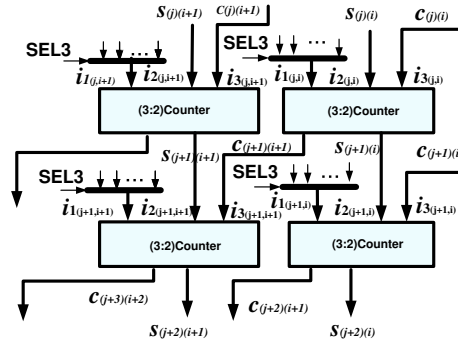


Figure 3.4: Detail organization into the AUSM extended array

Distribution of the processing terms: The multiplexer - SEL3 signal

A 8-to-1 multiplexer is used to route the data for the different proposed operations. The control signal SEL3 drives these multiplexers and controls that the (3:2)counter are fed with correct data. Multiplexer selector (SEL3) is a signal with the same function as SEL1 and SEL2, presented and explained in Chapter 2. In this case, SEL3 is a three-bit multiplexer selection signal. The encoding scheme is presented in Table 3.2.

Table 3.2: Selector signal SEL3 - Multiplexer behavior

Selector signal	Function
$SEL3 \equiv 000_2$	It is used to select the SAD operands.
$SEL3 \equiv 001_2$	Route partial products for unsigned multiplication operations.
$SEL3 \equiv 010_2$	Choose the integer signed-magnitude multiplication data.
$SEL3 \equiv 011_2$	It is used to select the integer two's complement values.
$SEL3 \equiv 100_2$	Choose the MAC operands (the partial products and the $w_{(0,i)}$ operand)
$SEL3 \equiv 101_2$	It is used to select the unsigned fractional values for multiplication.
$SEL3 \equiv 110_2$	Used to choose the signed magnitude fractional values.
$SEL3 \equiv 111_2$	Used to select the fractional two's complement values.

Table 3.3 summarizes the (3:2)counter input $i1_{(j,i)}$. The possible inputs in each 10 main *sections* of the AUSM scheme are abbreviated in column one of Table 3.3 as *Sec. 1*, *Sec. 2* and so on. The data values for $i1_{(j,i)}$ are selected using the 8-to-1 multiplexers with the use of control signal SEL3 for the different functionalities as described above in Table 3.2. From Table 3.3 it is evident that we process two's complement numbers with sign extension. Signed magnitude numbers are processed by the same hardware as positive numbers are.

Their sign bits are forced to zero and the result is updated with the *XOR* of multiplicand and multiplier signs like in the AUSM scheme.

Table 3.3: $i1_{(j,i)}$ inputs: The 8-to-1 multiplexer

	SAD	Unsigned Integer MUL	Signed Integer MUL	Two's Integer MUL	MAC Two's	Unsigned Fractional MUL	Signed Fractional MUL	Two's Fractional MUL
	000 ₂	001 ₂	010 ₂	011 ₂	100 ₂	101 ₂	110 ₂	111 ₂
Sec. 1	$in_{(j,i)}$	0	0	$z_{(j,15)}$	$z_{(j,15)}$	$f_{(j,i)}$	$f_{(j,i)}$	$f_{(j,i)}$
Sec. 2	$in_{(j,i)}$	0	0	$z_{(j,15)}$	$z_{(j,15)}$	0	0	$f_{(j,15)}$
Sec. 3	$in_{(j,i)}$	$z_{(j,i)}$	$z_{(j,i)}$	$z_{(j,i)}$	$z_{(j,i)}$	$f_{(j,i)}$	$f_{(j,i)}$	$f_{(j,i)}$
Sec. 4	$in_{(j,i)}$	0	0	$z_{(j,15)}$	$z_{(j,15)}$	0	0	$f_{(j,15)}$
Sec. 5	$in_{(j,i)}$	$z_{(j,i)}$	$z_{(j,i)}$	$z_{(j,i)}$	$z_{(j,i)}$	0	0	0
Sec. 6	$in_{(j,i)}$	$z_{(j,i)}$	$z_{(j,i)}$	$z_{(j,i)}$	$z_{(j,i)}$	$f_{(j,i)}$	$f_{(j,i)}$	$f_{(j,i)}$
Sec. 7	$in_{(j,i)}$	0	0	0	0	0	0	0
Sec. 8	$in_{(j,i)}$	0	0	0	0	$f_{(j,i)}$	$f_{(j,i)}$	$f_{(j,i)}$
Sec. 9	0	$z_{(15,i)}$	$z_{(15,i)}$	$z_{(15,i)}$	$z_{(15,i)}$	0	0	0
Sec. 10	0	0	0	0	$w_{(0,i)}$	0	0	0

Additionally, we notice that in the limit of both sides, left and right of the counter array, the carry input $i3_{(j,i)}$ uses an additional multiplexer which is in charge of propagating or inhibiting the carry out of the (3:2)counters of the right side. Also this multiplexer is used to introduce a Hot One (HO) or a zero. The HO is introduced when a two's complement subtraction operation is carried out in the SAD operations. Furthermore, we have to notice that the first row of (3:2)counters of both sides, the left and the right, use three multiplexers instead of one as described further (see the equations in section 3.2 for more details).

3.2 The Array Construction - Equations Description

The multiple operation array is composed by 32 columns and 16 rows of (3:2)counters, giving a total of 496 (3:2)counters. Figure 3.5 contains all the details presented in the scheme depicted in Figure 3.2(a). The notation used for representing (3:2)counters gives us information of the different kind of data received by these core logic blocks. The first columns on each side of the array, sections 3 and 7 in Figure 3.2(a) are used for introducing a Hot One (columns 0 and 16 and rows 1 to 8 of Figure 3.5 are used for that). The

rest of the column introduces zeros. These corrections are necessary for SAD calculation because of the following property: $A - B = A + \overline{B} + 1$, that holds true for two's complement. Also we notice that column 16 inhibits the carry propagation from the right side. It should also be noted that the last row of this figure, denoted by "+", represents the final stage CPA used to calculate the final outcome values of the (3:2)counter array.

The remainder of this section describes in details the mathematical equations used in the 10 sections of (3:2)counter array. We have detailed for all cases the input $i1_{(j,i)}$ and also the other equations for inputs $i2_{(j,i)}$ and $i3_{(j,i)}$ when the structure differs from the functionality presented in Figure 3.2(b). The (3:2)counters produces two outputs $s_{(j,i)}$ that correspond to the SUM, and $c_{(j,i)}$ for the CARRY output; where those outputs are a function of their corresponding inputs ($[c_{(j,i)}, s_{(j,i)}] = f[i1_{(j,i)}, i2_{(j,i)}, i3_{(j,i)}]$).

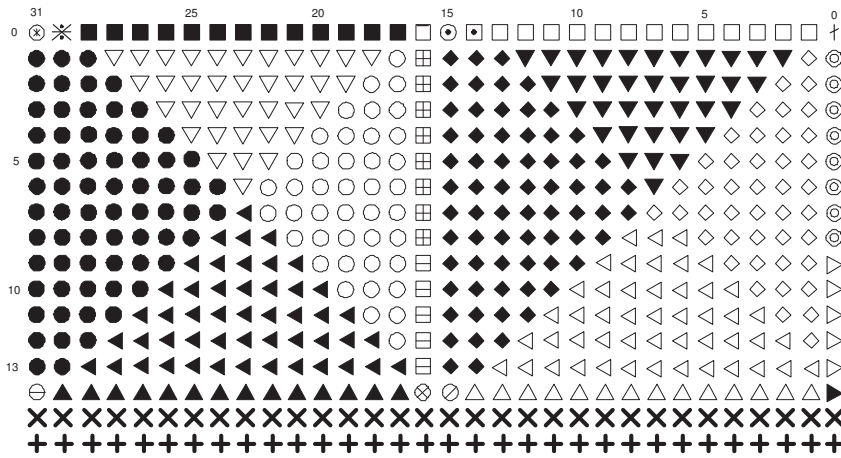


Figure 3.5: The AUSM extension - array organization

Consider the partial products $z_{(j,i)}$, $f_{(j,i)}$ and the SAD inputs (IN[32]), then the functionality of each AUSM section is described analytically below:

Section 1, is denoted by the symbols \square , \blacksquare , \ast , \otimes and ∇ . This section is used by the following operations: SAD, fractional multiplications in universal notations, integer multiplication in two's complement and MAC operations. The exact behavior of the different inputs is as follows:

□ :

$$\begin{aligned}
i1_{(0,16)} &= in_{(1,0)} \cdot (SEL3 \equiv "000") + z_{(1,15)} \cdot (SEL3 \equiv "001") + \\
&\quad z_{(1,15)} \cdot (SEL3 \equiv "010") + z_{(1,15)} \cdot (SEL3 \equiv "011") + \\
&\quad z_{(1,15)} \cdot (SEL3 \equiv "100") + f_{(15,1)} \cdot (SEL3 \equiv "101") + \\
&\quad f_{(15,1)} \cdot (SEL3 \equiv "110") + f_{(15,1)} \cdot (SEL3 \equiv "111") \\
i2_{(0,16)} &= in_{(2,0)} \cdot (SEL3 \equiv "000") + z_{(0,15)} \cdot (SEL3 \equiv "011") + \\
&\quad z_{(0,15)} \cdot (SEL3 \equiv "100") + f_{(14,2)} \cdot (SEL3 \equiv "101") + \\
&\quad f_{(14,2)} \cdot (SEL3 \equiv "110") + f_{(14,2)} \cdot (SEL3 \equiv "111") \\
i3_{(0,16)} &= in_{(3,0)} \cdot (SEL3 \equiv "000")
\end{aligned}$$

■ :

for $i = 1..13$

$$\begin{aligned}
i1_{(0,i+16)} &= in_{(1,i)} \cdot (SEL3 \equiv "000") + z_{(0,15)} \cdot (SEL3 \equiv "011") + \\
&\quad z_{(0,15)} \cdot (SEL3 \equiv "100") + f_{(15,i+1)} \cdot (SEL3 \equiv "101") + \\
&\quad f_{(15,i+1)} \cdot (SEL3 \equiv "110") + f_{(15,i+1)} \cdot (SEL3 \equiv "111") \\
i2_{(0,i+16)} &= in_{(2,i)} \cdot (SEL3 \equiv "000") + z_{(1,15)} \cdot (SEL3 \equiv "011") + \\
&\quad z_{(1,15)} \cdot (SEL3 \equiv "100") + f_{(14,i+2)} \cdot (SEL3 \equiv "101") + \\
&\quad f_{(14,i+2)} \cdot (SEL3 \equiv "110") + f_{(14,i+2)} \cdot (SEL3 \equiv "111") \\
i3_{(0,i+16)} &= in_{(3,i)} \cdot (SEL3 \equiv "000")
\end{aligned}$$

end

※ :

$$\begin{aligned}
i1_{(0,30)} &= in_{(1,14)} \cdot (SEL3 \equiv "000") + z_{(1,15)} \cdot (SEL3 \equiv "011") + \\
&\quad z_{(1,15)} \cdot (SEL3 \equiv "100") + f_{(15,1)} \cdot (SEL3 \equiv "101") + \\
&\quad f_{(15,1)} \cdot (SEL3 \equiv "110") + f_{(15,1)} \cdot (SEL3 \equiv "111") \\
i2_{(0,30)} &= in_{(2,14)} \cdot (SEL3 \equiv "000") + z_{(0,15)} \cdot (SEL3 \equiv "011") + \\
&\quad z_{(0,15)} \cdot (SEL3 \equiv "100") + f_{(14,15)} \cdot (SEL3 \equiv "111") \\
i3_{(0,30)} &= in_{(3,14)} \cdot (SEL3 \equiv "000")
\end{aligned}$$

⊗ :

$$\begin{aligned}
i1_{(0,31)} &= in_{(1,15)} \cdot (SEL3 \equiv "000") + z_{(1,15)} \cdot (SEL3 \equiv "011") + \\
&\quad z_{(1,15)} \cdot (SEL3 \equiv "100") + f_{(15,1)} \cdot (SEL3 \equiv "111") \\
i2_{(0,31)} &= in_{(2,15)} \cdot (SEL3 \equiv "000") + z_{(0,15)} \cdot (SEL3 \equiv "011") + \\
&\quad z_{(0,15)} \cdot (SEL3 \equiv "100") + f_{(14,15)} \cdot (SEL3 \equiv "111") \\
i3_{(0,31)} &= in_{(3,15)} \cdot (SEL3 \equiv "000")
\end{aligned}$$

```

∇ :
for j = 1...6
  n = j + 1,
  m = 13 - j,
  for i = n...m

    
$$i1_{(j,i+16)} = in_{(j+3,i)} \cdot (SEL3 \equiv "000") + z_{(j+1,15)} \cdot (SEL3 \equiv "011") +$$


$$z_{(j+1,15)} \cdot (SEL3 \equiv "100") + f_{(14-j,i+3)} \cdot (SEL3 \equiv "101") +$$


$$f_{(14-j,i+3)} \cdot (SEL3 \equiv "110") + f_{(14-j,i+3)} \cdot (SEL3 \equiv "111")$$


  end
end

```

Section 2, is denoted by \bullet . This *section* is active during the following operations: SAD, integer and fractional multiplication in two's complement and MAC operations. Its precise behavior is as follows:

```

• :
for j = 1...6
  n = 14 - j,
  for i = n...15

    
$$i1_{(j,i+16)} = in_{(j+3,i)} \cdot (SEL3 \equiv "000") + z_{(j+1,15)} \cdot (SEL3 \equiv "011") +$$


$$z_{(j+1,15)} \cdot (SEL3 \equiv "100") + f_{(14-j,15)} \cdot (SEL3 \equiv "111")$$


  end
end

```

```

• :
for j = 7...13
  n = j + 1,
  for i = n...15

    
$$i1_{(j,i+16)} = in_{(j+3,i)} \cdot (SEL3 \equiv "000") + z_{(j+1,15)} \cdot (SEL3 \equiv "011") +$$


$$z_{(j+1,15)} \cdot (SEL3 \equiv "100") + f_{(14-j,15)} \cdot (SEL3 \equiv "111")$$


  end
end

```

Section 3, is denoted by the symbols \boxplus , \boxminus , and \bigcirc . This *section* is active in all the operations: SAD, fractional and integer multiplication in universal notations, and MAC operations. In more details, the behavior is:

⊞ :

for $i = 1 \dots 8$

$$\begin{aligned}
 i1_{(j,16)} &= in_{(j+3,0)} \cdot (SEL3 \equiv "000") + z_{(j+1,15-j)} \cdot (SEL3 \equiv "001") + \\
 & z_{(j+1,15-j)} \cdot (SEL3 \equiv "010") + z_{(j,16-j)} \cdot (SEL3 \equiv "011") + \\
 & z_{(j,16-j)} \cdot (SEL3 \equiv "100") + f_{(14-j,j+2)} \cdot (SEL3 \equiv "101") + \\
 & f_{(14-j,j+2)} \cdot (SEL3 \equiv "110") + f_{(14-j,j+2)} \cdot (SEL3 \equiv "111") \\
 i3_{(j,16)} &= 1 \cdot (SEL3 \equiv "000") + c_{(j-1,i+15)} \cdot (SEL3 \equiv "001") + \\
 & c_{(j,i+16)} \cdot (SEL3 \equiv "010") + c_{(j,i+16)} \cdot (SEL3 \equiv "011") + \\
 & c_{(j,i+16)} \cdot (SEL3 \equiv "100") + c_{(j,i+16)} \cdot (SEL3 \equiv "101") + \\
 & c_{(j,i+16)} \cdot (SEL3 \equiv "110") + c_{(j,i+16)} \cdot (SEL3 \equiv "111")
 \end{aligned}$$

end

⊞ :

for $j = 9 \dots 13$

$$\begin{aligned}
 i1_{(j,16)} &= in_{(j+3,0)} \cdot (SEL3 \equiv "000") + z_{(j+1,15-j)} \cdot (SEL3 \equiv "001") + \\
 & z_{(j+1,15-j)} \cdot (SEL3 \equiv "010") + z_{(j,15-j)} \cdot (SEL3 \equiv "011") + \\
 & z_{(j,15-j)} \cdot (SEL3 \equiv "100") + f_{(14-j,j+2)} \cdot (SEL3 \equiv "101") + \\
 & f_{(14-j,j+2)} \cdot (SEL3 \equiv "110") + f_{(14-j,j+2)} \cdot (SEL3 \equiv "111") \\
 i3_{(j,16)} &= 1 \cdot (SEL3 \equiv "000") + c_{(j,i+16)} \cdot (SEL3 \equiv "001") + \\
 & c_{(j,i+16)} \cdot (SEL3 \equiv "010") + c_{(j,i+16)} \cdot (SEL3 \equiv "011") + \\
 & c_{(j,i+16)} \cdot (SEL3 \equiv "100") + c_{(j,i+16)} \cdot (SEL3 \equiv "101") + \\
 & c_{(j,i+16)} \cdot (SEL3 \equiv "110") + c_{(j,i+16)} \cdot (SEL3 \equiv "111")
 \end{aligned}$$

end

○ :

for $j = 1 \dots 6$

m = j,

for $i = 1 \dots m$

$$\begin{aligned}
 i1_{(j,i+16)} &= in_{(j+3,i)} \cdot (SEL3 \equiv "000") + z_{(j+1,i+14)} \cdot (SEL3 \equiv "001") + \\
 & z_{(j+1,i+14)} \cdot (SEL3 \equiv "010") + z_{(j+1,i+14)} \cdot (SEL3 \equiv "011") + \\
 & z_{(j+1,i+14)} \cdot (SEL3 \equiv "100") + f_{(14-j,i+3)} \cdot (SEL3 \equiv "101") + \\
 & f_{(14-j,i+3)} \cdot (SEL3 \equiv "110") + f_{(14-j,i+3)} \cdot (SEL3 \equiv "111")
 \end{aligned}$$

end

end

○ :
for $j = 7 \dots 12$
 $m = 13 - j,$
 for $i = 1 \dots m$

$$\begin{aligned} i1_{(j,i+16)} = & in_{(j+3,i)} \cdot (SEL3 \equiv \text{"000"}) + z_{(j+1,i+15-j)} \cdot (SEL3 \equiv \text{"001"}) + \\ & z_{(j+1,i+15-j)} \cdot (SEL3 \equiv \text{"010"}) + z_{(j+1,i+15-j)} \cdot (SEL3 \equiv \text{"011"}) + \\ & z_{(j+1,i+15-j)} \cdot (SEL3 \equiv \text{"100"}) + f_{(14-j,i+j+2)} \cdot (SEL3 \equiv \text{"101"}) + \\ & f_{(14-j,i+j+2)} \cdot (SEL3 \equiv \text{"110"}) + f_{(14-j,i+j+2)} \cdot (SEL3 \equiv \text{"111"}) \end{aligned}$$

end

end

Section 4, is denoted by ◀. This *section* is active during the following operations: SAD, fractional and integer multiplication in two's complement and MAC operations. Its precise behavior is as follows:

◀ :
for $j = 7 \dots 13$
 $n = 14 - j,$
 $m = j,$
 for $i = n \dots m$

$$\begin{aligned} i1_{(j,i+16)} = & in_{(j+3,i)} \cdot (SEL3 \equiv \text{"000"}) + z_{(j+1,15-j+i)} \cdot (SEL3 \equiv \text{"001"}) + \\ & z_{(j+1,15-j+i)} \cdot (SEL3 \equiv \text{"010"}) + z_{(j+1,15-j+i)} \cdot (SEL3 \equiv \text{"011"}) + \\ & z_{(j+16,15-j+i)} \cdot (SEL3 \equiv \text{"100"}) + f_{(14-j,15)} \cdot (SEL3 \equiv \text{"111"}) \end{aligned}$$

end

end

Section 5, is denoted by the symbols †, □, ◻, ⊙ and ∇. This *section* is active during the following operations: SAD, integer multiplication in universal notations and MAC operations. In more details, the behavior is:

† :

$$\begin{aligned} i1_{(0,0)} = & in_{(17,0)} \cdot (SEL3 \equiv \text{"000"}) + z_{(0,0)} \cdot (SEL3 \equiv \text{"001"}) + \\ & z_{(0,0)} \cdot (SEL3 \equiv \text{"010"}) + z_{(0,0)} \cdot (SEL3 \equiv \text{"011"}) + \\ & z_{(0,0)} \cdot (SEL3 \equiv \text{"100"}) \\ i2_{(0,0)} = & in_{(18,0)} \cdot (SEL3 \equiv \text{"000"}) \\ i3_{(0,0)} = & in_{(19,0)} \cdot (SEL3 \equiv \text{"000"}) \end{aligned}$$

□ :

for $i = 1..13$

$$i1_{(0,i)} = in_{(17,i)} \cdot (SEL3 \equiv "000") + z_{(0,i)} \cdot (SEL3 \equiv "001") + \\ z_{(0,i)} \cdot (SEL3 \equiv "010") + z_{(0,i)} \cdot (SEL3 \equiv "011") + \\ z_{(0,i)} \cdot (SEL3 \equiv "100")$$

$$i2_{(0,i)} = in_{(18,i)} \cdot (SEL3 \equiv "000") + z_{(1,i-1)} \cdot (SEL3 \equiv "001") + \\ z_{(1,i-1)} \cdot (SEL3 \equiv "010") + z_{(1,i-1)} \cdot (SEL3 \equiv "011") + \\ z_{(1,i-1)} \cdot (SEL3 \equiv "100")$$

$$i3_{(0,i)} = in_{(19,i)} \cdot (SEL3 \equiv "000")$$

end

□ :

$$i1_{(0,14)} = in_{(17,14)} \cdot (SEL3 \equiv "000") + z_{(0,14)} \cdot (SEL3 \equiv "001") + \\ z_{(0,14)} \cdot (SEL3 \equiv "010") + z_{(0,14)} \cdot (SEL3 \equiv "011") + \\ z_{(0,14)} \cdot (SEL3 \equiv "100") + f_{(14,0)} \cdot (SEL3 \equiv "101") + \\ f_{(14,0)} \cdot (SEL3 \equiv "110") + f_{(14,0)} \cdot (SEL3 \equiv "111")$$

$$i2_{(0,14)} = in_{(18,14)} \cdot (SEL3 \equiv "000") + z_{(1,13)} \cdot (SEL3 \equiv "001") + \\ z_{(1,13)} \cdot (SEL3 \equiv "010") + z_{(1,13)} \cdot (SEL3 \equiv "011") + \\ z_{(1,13)} \cdot (SEL3 \equiv "100")$$

$$i3_{(0,14)} = in_{(19,14)} \cdot (SEL3 \equiv "000")$$

⊙ :

$$i1_{(0,15)} = in_{(17,15)} \cdot (SEL3 \equiv "000") + z_{(0,15)} \cdot (SEL3 \equiv "001") + \\ z_{(0,15)} \cdot (SEL3 \equiv "010") + z_{(0,15)} \cdot (SEL3 \equiv "011") + \\ z_{(0,15)} \cdot (SEL3 \equiv "100") + f_{(15,0)} \cdot (SEL3 \equiv "101") + \\ f_{(15,0)} \cdot (SEL3 \equiv "110") + f_{(15,0)} \cdot (SEL3 \equiv "111")$$

$$i2_{(0,15)} = in_{(18,15)} \cdot (SEL3 \equiv "000") + z_{(1,14)} \cdot (SEL3 \equiv "001") + \\ z_{(1,14)} \cdot (SEL3 \equiv "010") + z_{(1,14)} \cdot (SEL3 \equiv "011") + \\ z_{(1,14)} \cdot (SEL3 \equiv "100") + f_{(14,1)} \cdot (SEL3 \equiv "101") + \\ f_{(14,1)} \cdot (SEL3 \equiv "110") + f_{(14,1)} \cdot (SEL3 \equiv "111")$$

$$i3_{(0,15)} = in_{(19,15)} \cdot (SEL3 \equiv "000") + HO \cdot (SEL3 \equiv "011") + \\ HO \cdot (SEL3 \equiv "100") + HO \cdot (SEL3 \equiv "111")$$

▽ :

for $j = 1..6$ $n = j + 1,$ $m = 13 - j,$ **for** $i = n..m$

$$i1_{(j,i+16)} = in_{(j+19,i)} \cdot (SEL3 \equiv "000") + z_{(j+1,i-j-1)} \cdot (SEL3 \equiv "001") + \\ z_{(j+1,i-j-1)} \cdot (SEL3 \equiv "010") + z_{(j+1,i-j-1)} \cdot (SEL3 \equiv "011") + \\ z_{(j+1,i-j-1)} \cdot (SEL3 \equiv "100")$$

end

end

Section 6, is denoted by the symbol \bullet . This *section* is active in all the operations: SAD, fractional and integer multiplication in universal notations and MAC operations. Its precise behavior is as follows:

\bullet :

for $j = 1..6$

n = 14 - j,

for $i = n..15$

$$\begin{aligned} i1_{(j,i)} = & in_{(j+19,i)} \cdot (SEL3 \equiv "000") + z_{(j+1,i-j-1)} \cdot (SEL3 \equiv "001")+ \\ & z_{(j+1,i-j-1)} \cdot (SEL3 \equiv "010") + z_{(j+1,i-j-1)} \cdot (SEL3 \equiv "011")+ \\ & z_{(j+1,i-j-1)} \cdot (SEL3 \equiv "100") + f_{(14-j,i-14+j)} \cdot (SEL3 \equiv "101")+ \\ & f_{(14-j,i-14+j)} \cdot (SEL3 \equiv "110") + f_{(14-j,i-14+j)} \cdot (SEL3 \equiv "111") \end{aligned}$$

end

end

\bullet :

for $j = 7..13$

n = 15 - j,

for $i = n..15$

$$\begin{aligned} i1_{(j,i)} = & in_{(j+19,i)} \cdot (SEL3 \equiv "000") + z_{(j+1,i+j-1)} \cdot (SEL3 \equiv "001")+ \\ & z_{(j+1,i+j-1)} \cdot (SEL3 \equiv "010") + z_{(j+1,i+j-1)} \cdot (SEL3 \equiv "011")+ \\ & z_{(j+1,i+j-1)} \cdot (SEL3 \equiv "100") + f_{(14-j,i-14+j)} \cdot (SEL3 \equiv "101")+ \\ & f_{(14-j,i-14+j)} \cdot (SEL3 \equiv "110") + f_{(14-j,i-14+j)} \cdot (SEL3 \equiv "111") \end{aligned}$$

end

end

Section 7, is denoted by the symbols \odot , \triangleright and \diamond . This *section* is active in SAD operations. In more details, the behavior is:

\odot :

for $j = 1..8$

$$i1_{(j,0)} = in_{(j+19,0)} \cdot (SEL3 \equiv "000")$$

$$i3_{(j,0)} = 1 \cdot (SEL3 \equiv "000")$$

end

▷ :

for $j = 9..13$

$$i1_{(j,0)} = in_{(j+19,0)} \cdot (SEL3 \equiv "000")$$

$$i3_{(j,0)} = 0$$

end

◇ :

for $j = 1..7$

m = j,

for $i = 1..m$

$$i1_{(j,i+16)} = in_{(j+19,i)} \cdot (SEL3 \equiv "000")$$

end

end

◇ :

for $j = 8..12$

m = 14 - j,

for $i = 1..m$

$$i1_{(j,i)} = in_{(j+19,i)} \cdot (SEL3 \equiv "000")$$

end

end

Section 8, is denoted by ◁. This *section* is active in SAD and fractional multiplication in universal notations. Its precise behavior is as follows:

◁ :

for $j = 9..13$

n = 16 - j,

m = j-1,

for $i = n..m$

$$i1_{(j,i)} = in_{(j+17,i)} \cdot (SEL3 \equiv "000") + f_{(16-j,16-j-i)} \cdot (SEL3 \equiv "101") +$$

$$f_{(16-j,16-j-i)} \cdot (SEL3 \equiv "110") + f_{(16-j,16-j-i)} \cdot (SEL3 \equiv "111")$$

end

end

Section 9, is denoted by ►, △, ⊗, ⊗, ▲ and ⊖. This *section* is active in integer multiplication in universal notations and MAC operations. Its precise behavior is as follows:

► :

$$i1_{(14,0)} = f_{(0,0)} \cdot (SEL3 \equiv "101") + f_{(0,0)} \cdot (SEL3 \equiv "110") + f_{(0,0)} \cdot (SEL3 \equiv "111")$$

△ :

for $i = 1..14$

$$i1_{(14,i)} = f_{(0,i)} \cdot (SEL3 \equiv "101") + f_{(0,i)} \cdot (SEL3 \equiv "110") + f_{(0,i)} \cdot (SEL3 \equiv "111")$$

end

⊙ :

$$i1_{(14,15)} = z_{(15,0)} \cdot (SEL3 \equiv "001") + z_{(15,0)} \cdot (SEL3 \equiv "010") + z_{(15,0)} \cdot (SEL3 \equiv "011") + z_{(15,0)} \cdot (SEL3 \equiv "100") + f_{(0,15)} \cdot (SEL3 \equiv "101") + f_{(0,15)} \cdot (SEL3 \equiv "110") + f_{(0,15)} \cdot (SEL3 \equiv "111")$$

⊗ :

$$i1_{(14,16)} = z_{(15,1)} \cdot (SEL3 \equiv "001") + z_{(15,1)} \cdot (SEL3 \equiv "010") + z_{(15,1)} \cdot (SEL3 \equiv "011") + z_{(15,1)} \cdot (SEL3 \equiv "100") + f_{(0,15)} \cdot (SEL3 \equiv "101") + f_{(0,15)} \cdot (SEL3 \equiv "110") + f_{(0,15)} \cdot (SEL3 \equiv "111")$$

▲ :

for $i = 1..14$

$$i1_{(14,i+16)} = z_{(15,i+1)} \cdot (SEL3 \equiv "001") + z_{(15,i+1)} \cdot (SEL3 \equiv "010") + z_{(15,i+1)} \cdot (SEL3 \equiv "011") + z_{(15,i+1)} \cdot (SEL3 \equiv "100") + f_{(0,15)} \cdot (SEL3 \equiv "111")$$

end

⊖ :

$$i1_{(14,31)} = z_{(15,15)} \cdot (SEL3 \equiv "011") + z_{(15,15)} \cdot (SEL3 \equiv "100") + f_{(0,15)} \cdot (SEL3 \equiv "111")$$

Section 10, is denoted by \times . This *section* is active in MAC operations. Its behavior is as follows:

for $i = 0..31$

$$i1_{(15,i)} = w(0,i) \cdot (SEL3 \equiv "100")$$

end

3.3 Experimental Results and Analysis

The AUSM extension and the necessary control logic including the *carry unit* has been implemented using VHDL. We have synthesized, functionally tested, and validated with the ISE 5.2 Xilinx environment [57] and Modelsim [58] respectively, for the VIRTEX II PRO xc2vp100-7ff1696 FPGA device. The AUSM extended design has the following embedded units and features:

- A 16×16 bit multiplier for integer and fractional representations in universal notations;
- 4×4 SAD unit operating in universal notation, doubling the processing capacity of the original AUSM;
- The Multiply-Accumulation (MAC) operation for two's complement notation.

The individual units such as: unsigned multiplier; signed magnitude multiplier; two's complement multiplier; two rectangular SAD units (2 half of the rectangular array); and MAC unit has been implemented in stand alone units, without synthesizing the overhead logic of the whole collapsed unit. Previously presented units like the AUSM [5] for integer numbers and Baugh and Wooley (B&W) signed multiplier [59] are synthesized with the same ISE toolchain in order to have fair comparison with our proposal. Table 3.4 summarizes the performance in terms of delay for all structures we investigated.

The additional logic introduced into the extended AUSM reduces the performance of the multiply and SAD functionalities as reported in Table 3.4 with respect to the latencies achieved with the AUSM unit (see Table 2.4 in Chapter 2). Our proposal, compared with the embedded (collapsed) presents an extra delay. It is around 12 % for a 16 bits MAC operand, and up to 50 % for a 32 bits MAC compared to the previous single functionality units such as Baugh-Wooley or a simple integer AUSM. This increased latency diminishes to 27 % when the fast carry logic provided in the Xilinx FPGAs is used [60] in the final stage adder. The additional latency introduced by the AUSM-extended scheme when it is compared with the previous AUSM scheme is due to two main factors: the first relates to the multiplexer used to provide the data into the input $i1_{(j,i)}$, which presents a constant delay for all the logic blocks; the second is due to the additional multiplexer introduced in the array to separate logically the right and left sides.

Table 3.4: AUSM extension and related units
- Latency -

Unit	Logic Delay (ns)	Wire Delay (ns)	Total Delay (ns)
SAD ‡	13.184	12.006	25.191
Unsigned Multiplier ‡	14.589	14.639	29.282
Two's I ‡	12.595	15.564	28.159
Two's F ‡	15.153	16.825	31.978
Baugh&Wooley ‡	15.555	15.826	31.381
U-SAD-M ‡	15.877	15.741	31.618
U-SAD-M †	14.311	9.568	23.879
MAC ‡	15.062	19.064	34.662
Our Proposal-32 MAC ‡	21.351	26.040	47.391
Our Proposal-16 MAC ‡	16.521	19.065	35.586
Our Proposal-32 MAC †	15.311	15.127	30.438
Carry Unit	2.576	3.338	5.914

RCA final adder: ‡ : LUT based ; † : Based on Xilinx Fast Carry Logic. [60]

Instead of using a regular array, a Wallace tree [63] can be implemented in order to accelerate the performance of the operations. By using Wallace trees, the critical path is reduced from 16 CSA down to 6 CSA delays. This is an improvement of 62 % in the total (3:2)counter array delay.

Concerning the silicon used by the extended AUSM scheme and the other structures, depicted on Table 3.5, the hardware overhead of the presented unit is considerable and is the price we pay for its multi-functional characteristic. Nevertheless, if the eight units are implemented separately, we will need two times the hardware resources. We should also consider that 6/8 of the basic logic block array is shared among the eight operations. Therefore, these operations are good candidates for partial reconfiguration implementation, exploiting differences of the functional units.

3.4 Conclusions

We have presented an extended AUSM array organization. The proposed unit can be implemented on a VLSI circuit as a programmable unit, and it can also be used in a reconfigurable technology as a run-time reconfigurable unit. The whole array is configured using multiplexers, which can be replaced by fast connections on partially reconfigurable hardware. Several units such as:

Table 3.5: AUSM extension and related units
- Hardware Use -

Unit	# Slices	# LUTs	# IOBs
SAD †	242	421	272
Unsigned Multiplier ‡	300	524	64
Two's I ‡	294	511	64
Two's F ‡	443	770	64
Baugh&Wooley ‡	330	574	65
AUSM ‡	686	1198	322
AUSM †	658	1170	322
MAC ‡	358	622	96
Our Proposal ‡	1373	2492	643
Our Proposal-16 MAC ‡	1360	2465	643
Our Proposal †	1354	2458	643
Carry Unit	35	61	64

RCA final adder: ‡ : LUT based ; † : Based on Xilinx Fast Carry Logic [60].

multiplication in integer and fractional notations; the sum of absolute differences in unsigned, signed magnitude and two's complement notations, a multiply-accumulation (MAC) unit for two's complement representation and Baugh&Wooley multiplier were coded and synthesized. Their performance and complexity has been compared with our proposal. A brief analysis of the obtained results in terms of used area and time delay suggests a maximum working frequency of 27.5 MHz for the calculation of 4×4 SAD macro-block and 32.85 MHz for MAC. The same delay was obtained for the other multiplier operations in a VIRTEX II PRO device using a 3% of the available slices of the chosen FPGA.

Chapter 4

Fixed Point Dense and Sparse Matrix-Vector Multiply Arithmetic Unit

Matrix multiplication/addition is an important operation in both scientific and media applications (e.g. 3D graphics, realistic video games). The main differences between the two domains are that media applications typically use short, fixed point data formats, while in scientific applications, floating-point data are considered. There are numerous approaches for matrix hardware multiplication using systolic array structures (see for example [64–68]). These approaches consider either sparse or dense matrices computation, and mainly floating point operations [69–71]. In this chapter, we consider typical media data formats and propose a fixed point arithmetic unit for the collapsing of both sparse and dense computations. That is, both dense and sparse matrix computations are performed using the same unit. We introduce a simple numeric expression of this operation and we collapsed the necessary hardware unit capable to process in parallel several operations for the inner loop of a matrix-vector multiply operation. In this chapter:

- We propose a single unit for 16-bit fixed-point data, incorporating 4 sub-units. The design computes 4 multiplications and up to 12 additions, operating upon dense and sparse matrix formats. Sparse matrix data are compressed into small dense sub matrices;

- We introduce a simple algorithm to schedule the incoming data into the available resources. Our algorithm targets for maximum throughput and allocates the data entries into:
 - Four identical (16:2) reduction-trees¹, planed to be used as four embedded multiply units.
 - Four different reduction trees: (12:2), (10:2), (8:2) and (6:2) used for multiple-operand addition of the products and previous results when those are available.
 - Four (2:1) reduction units used to compute the outcomes of the multi-operand compression trees.
- We propose a 4 stage pipelined architecture that can run on a Virtex II PRO at 120 MHz reaching a throughput of 1.9 GOPS. Potentially, our design can reach performance of 21 GOPS on a larger xc2vp100-6 FPGA device, with 11 fixed-point dense and sparse matrix-vector-multiply units.

The remainder of this chapter is organized as follows. Section 4.1 outlines the sparse matrix compression formats we consider. Section 4.2 extensively describes the proposed unit and presents the 4 stage pipeline design. Section 4.3 describes the prototype and experimental results. Finally, the chapter concludes with Section 4.4.

4.1 Background

Let us define an $n \times n$ matrix $A[N][N] = [A_{i,j}]_{i,j=0,1,\dots,n-1}$, a vector $b[N] = [b_i]_{i=0,1,\dots,n-1}$, and the result of the matrix by vector multiplication $c[N] = [c_i]_{i=0,1,\dots,n-1}$. Mathematically, the matrix by vector multiplication operation is described as follows:

$$c = A \times b \tag{4.1}$$

with

$$c_i = \sum_{k=0}^{n-1} a_{i,k} b_k \quad i = 0, 1, \dots, n - 1 \tag{4.2}$$

¹Trees are very fast structures for summing partial products. In a tree, counters and compressors are arranged in different parallel connections to accelerate the multiple operand addition.

Numerous hardware mechanisms have been developed for efficient implementation of equation (4.2). Solutions include vector processors and SIMD multimedia instructions to deal with linear arrays and vectors [67, 72, 73]. Different numbers of parallel units are used to compute the inner loop of the equation (4.2), and in the case of sparse computations they are intended to work with one of several sparse matrix vector formats [74, 75]. In this chapter, we consider three formats for sparse matrix representations described in detail in the subsections to follow.

4.1.1 Sparse Matrix Compression Formats

We begin presenting the Compressed Row Storage as a good example of the most popular format and describe two improved formats denoted as the Block Based Compression Storage and the Hierarchical Sparse Matrix. The formats are described with the help of Figure 4.1, which depicts a 32×32 matrix $A[N][N]$ with several nonzero elements represented by numbers or by an “x”. The dense vector b and the resulting vector c are also illustrated. The partitioning of matrix A visualizes better the sparse formats we consider later in this section.

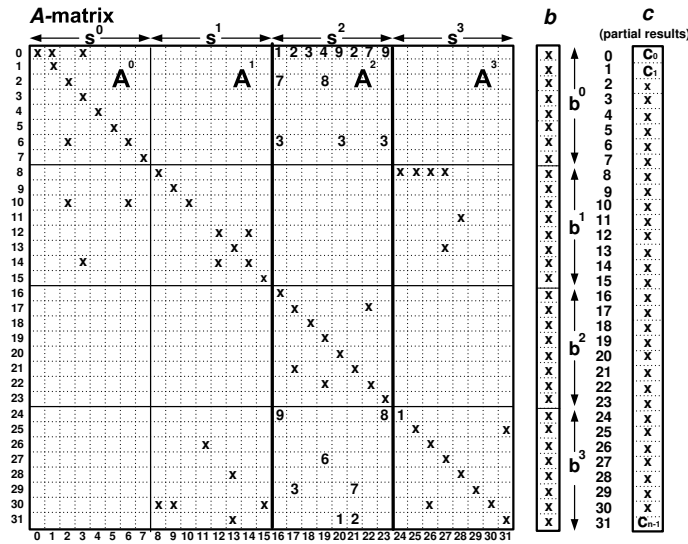


Figure 4.1: Sparse matrix representation

The Compressed Row Storage (CRS)

Figure 4.2 illustrates the CRS representation of a small sparse matrix with 7 nonzero elements. In this format, the compression is achieved using a linear array A_N to store nonzero elements of matrix A in a row-wise way. A second linear array A_J , is used to store the column index for each nonzero element and a third one, A_I , holds the indices of the first element of a row in A_N (see e.g. [68,76,77]).

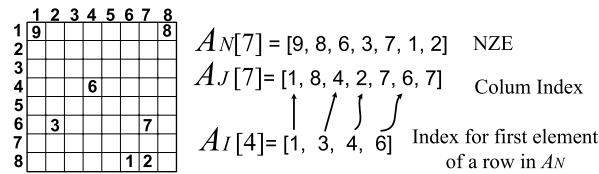


Figure 4.2: Compressed Row Storage (CRS) format

The Block Based Compression Storage (BBCS)

In the BBCS format [68], the $n \times n$ A matrix is fractionated in $\lceil \frac{n}{s} \rceil$ Vertical Blocks (VBs), where s corresponds to a processing section, as the 4 VBs presented in Figure 4.1. Figure 4.3 presents the BBCS format for nonzero values. The format enables us to process matrix A in sections A_m for $m = 0, 1, 2, 3, \dots, \lceil \frac{n}{s} \rceil - 1$; in the same way the vector b can be fractionated into $b_m = [b_{ms}, b_{ms+1}, \dots, b_{m.s+s-1}]$, where s is again the split section. Therefore, we are able to multiply each section A_m with the correspondent b_m section, segmenting in this way the matrix processing.

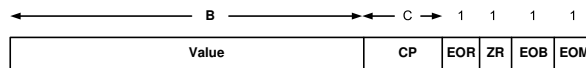


Figure 4.3: BBCS format

The fields presented in the BBCS format, and depicted in Figure 4.3, have the following interpretation:

- **Value:** Specifies the nonzero element of matrix A in a B-bits word representation when $ZR = 0$. When $ZR = 1$, this field denotes the number of subsequent block rows without NZE.

- **Column Position (CP):** Represents the column position for an element of matrix A within a vertical block A_m . For different blocks, the CP value is $j \bmod m$, represented in C-bits.
- **End-of-Row Flag (EOR):** This bit is asserted when the current nonzero value is the last element of the row, otherwise is zero.
- **Zero-Row-Flag (ZR):** Asserted when the current row has zero elements only.
- **End-of-block-Flag (EOB):** Indicates the last non zero value of the block.
- **End-of-Matrix-Flag (EOM):** Asserted when the particular nonzero element is the last one of the matrix.

Figure 4.4, presents an example of this format. We use a small sparse matrix extracted from the lowest part of Vertical Block A^2 depicted in Figure 4.1.

	1	2	3	4	5	6	7	8
1	9							8
2								
3								
4				6				
5								
6		3						7
7								
8					1	2		

	1	2	3	4	5	6
1	9	1	0	0	0	0
2	8	8	1	0	0	0
3	2	-	1	1	0	0
4	6	4	1	0	0	0
5	1	-	1	1	0	0
6	3	2	0	0	0	0
7	7	7	1	0	0	0
8	1	-	1	1	0	0
9	1	6	0	0	0	0
10	2	7	1	0	1	0

1	: Value
2	: Column Position
3	: EOR flag
4	: ZR flag
5	: EOB flag
6	: EOM flag

Figure 4.4: BBCS format example

The Hierarchical Sparse Matrix Format (HiSM)

In this hierarchical format, an $M \times N$ matrix A is partitioned in $\lceil \frac{M}{s} \rceil \times \lceil \frac{N}{s} \rceil$ squared sub-matrices. Figure 4.5 presents an example with 16 sub-matrices with $M = N = 32$, and section size $s = 8$. The nonzero values in this format, as well as the positional information combined, are stored in a row-wise array in memory. The format uses two indexing levels; the first one points to the square blocks (Level 1), which contains nonzero elements and the second

indexes the nonzero elements into the squared blocks (Level 0), using column and row representation [65].

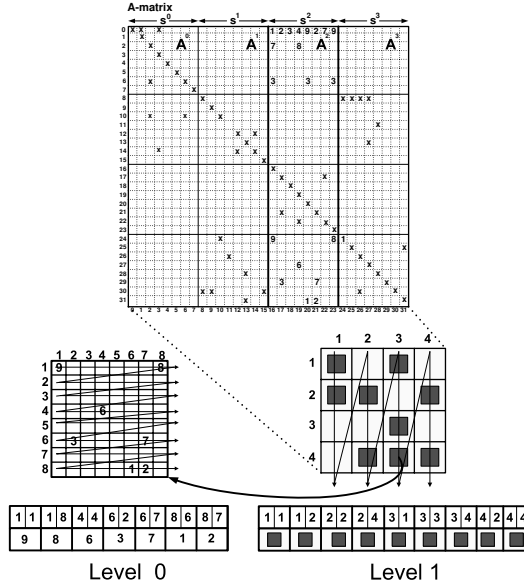


Figure 4.5: HiSM format example

A Common Hardware Structure for Processing Different Formats

Considering the aforementioned three compression formats, we propose a common hardware unit capable to process all of them after a specific pre-processing stage. The CRS can be translated into a BBCS representation with the use of simple hardware which decrements by one the values in the CRS A_I index, converting in this way the pointer of the first element of a row into a position associated with the EOR flag in the BBCS format.² Furthermore, the low level indexing in the HiSM (Level 0) can be modified (HiSM-M) to present a similar format to the BBCS, constituting in this way a square section version of the BBCS format. We denote the matrix A of such format by $A^{s \times s}$, where the exponent denotes a square matrix A ($s \times s$) of section size s . For this purpose we have to add an EOR flag to HiSM format. Summarizing, we propose a hardware which works with the nonzero elements, a column pointer as well as an EOR flag for the processing of the considered formats. Table 4.1

²Special attention has to be paid to the first element of the array.

presents in short our proposals. Column three identifies whether the particular format require a column index. Column four, whether a row index is required, and EOR - the type of end-of-row indication. In dense matrices and in HiSM-M, EOR flag is implicit, in BBCS- it is explicit, while CRC has to be extracted.

Table 4.1: Comparison of matrix-vector formats

Format	Value	Column Index	Row Index	EOR
Dense	$A[M][N]$	Yes	Yes	Implicit
CRS	$A_{(N)}[M]$	$A_{(J)}$	N.A.	Extracted
BBCS	$A_m[N][M]$	Yes	N.A.	Explicit
HiSM-M	$A^{s \times s}[S][S]$	Yes	N.A.	Implicit

Therefore, a block A_m where $m = 0, 1, \dots, \lceil \frac{n}{s} \rceil - 1$ and s is the processing section size, can be computed in parts. To achieve this, A_m has to be stored row-wise with increasing row number order. Thus, each A_m block will correspond to a section s of vector $b_m = [b_{ms}, b_{ms+1}, \dots, b_{ms+s-1}]$. So that each A_m can be multiplied by its corresponding b_m section of the vector. A hardware that exploits this characteristics intended to support the aforementioned formats is proposed in the next section.

4.2 Dense and Sparse Matrix-Vector Multiply Unit

Dense matrix-vector multiplication (DMVM) and sparse matrix-vector multiplication (SMVM) are easily parallelizable operations. Equation 4.3 suggests the potential for parallelism of four elements per cycle of any particular row.

$$c_i = \sum_{k=0}^3 a_{i,k} b_k + \sum_{k=4}^7 a_{i,k} b_k + \dots + \sum_{k=n-4}^{n-1} a_{i,k} b_k \quad (4.3)$$

According to equation 4.3, the computing of the dense matrix-vector multiply operation can be accelerated with a hardware that process concurrently s entries, in this case $s = 4$. Furthermore, the same hardware should be capable to deal with the sparse matrix-vector multiply operation, which contains a variable number of entries per row. This leads to consider situations in which the nonzero elements can belong to different rows and columns. Our hardware

solution uses dynamic channels for allocating the data into a hardware composed by four parallel execution units ($s = 4$). These units were designed in a collapsed way [6], optimized in terms of hardware complexity and delays. Our design was implemented in a 4-stage pipeline organization. A simple control algorithm is used to enable the hardware unit to support $n \times s$ sections, processing in this way section sizes like the ones described in Figure 4.1, as well as different cases³ like the one established by equation 4.2. The main tasks performed by our control allocation algorithm are:

1. Select one set of data (4 inputs) from a pool of two sets residing in the local memory;
2. Route the correct dense vector multiplicand and a dense or sparse matrix multiplier into a set of reduction trees;
3. Chose the necessary multi-operand units from a set of available resources for the addition of scalar multiplication results and a possible precalculated value. Depending on the incoming data, the algorithm allocates a certain number of multi-operand units, determined by the particular design ($s = 4$ in this case);
4. Add the final partial values into the final stage of (2:1) CPA adders, and save those results into the registers for write back into memory.

Memory Model: The memory model used to support the 4 stage pipeline fixed-point dense and sparse matrix-vector-multiply arithmetic accelerator unit denotes vector registers by **VR**, and Control Register by **CR**. Thus, our design has the following registers:

- **VR1:** 8 entry register file for 16-bit matrix A_m elements.
- **CR:** 8 entry register file for 8-bit index information of the nonzero elements of matrix A_m stored in VR1. Specifically we use the column index and the EOR flag.
- **VR2:** 8 entry register file for 16-bit dense vector $b_m[8]$.
- **VR3-O:** 8 entry register file for 32-bit $c_i[8]$ write-back.
- **VR3-I:** 8 entry register file for 32-bit $c_i[8]$, used to load the previous results from memory.

³Data can come from different rows, and in this proposal we present the processing of $2 \times s$ data.

4.2.1 The Pipeline Structure

The pipeline structure is divided into the following stages:

- Stage 1.** Vector Read.
- Stage 2.** Multiple reduction trees.
- Stage 3.** Multiple-Addition reduction trees.
- Stage 4.** Final Addition and result update.

In the following subsection we describe in detail each of the proposed pipeline stages:

Stage 1) Vector Read: Figure 4.6 describes schematically the functionality of this stage. The three main register files: VR1, VR2 and CR are actualized from memory through the “Memory Bus” (i.e. form embedded RAM). VR1, as stated before, stores the matrix elements A_m . VR2 is used to store vector b_m , while CR is used for hold the index information of the nonzero elements.

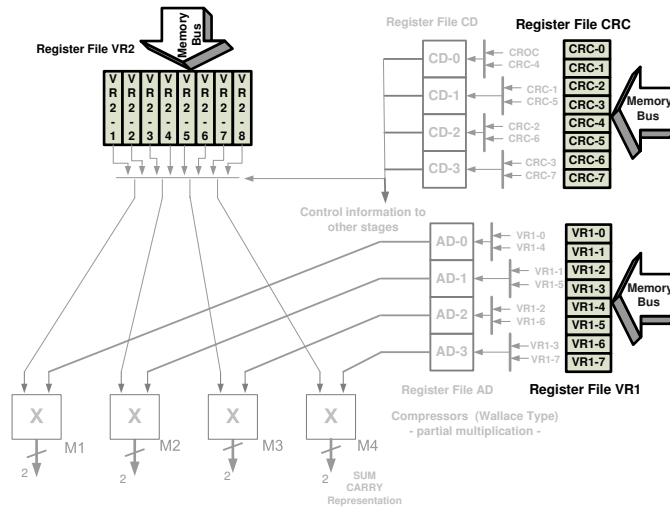


Figure 4.6: Vector read

Stage 2) Multiply reduction trees: In the second stage, the register files A-Buffer Data (AD) and Control Data (CD) are loaded. A toggle Flip-Flop

(FF) controls which quadruple set of VR1 and CR registers are routed with the eight 2-to-1 MUXs⁴ to the registers AD and CD respectively, as depicted in Figure 4.7 (a). Then the Column Positions (CPs) information held in CD registers, is used for controlling the four 8-to-1 muxes (MD) and route the VR2 values. At this point, a partial multiplication operation of the 4 matrix A elements and the corresponding dense vector b is carried out. Wallace type [63] trees without the final (2:1) addition are designed to compute the multiplication operations (see Figure 4.7 (b)), saving with this organization the unnecessary repetition of the addition operations. Our approach of collapsing several multiple operand additions and other related operations through the entire pipeline is the design philosophy adopted here. The final stage implements the necessary carry propagation adder. Using this approach, we save area and reduce at the same time the processing delay. We notice that Figure 4.7 (b) does not specify the register-pipeline (R-Pipeline) that stores tree compressor results (SUMs and CARRYs).

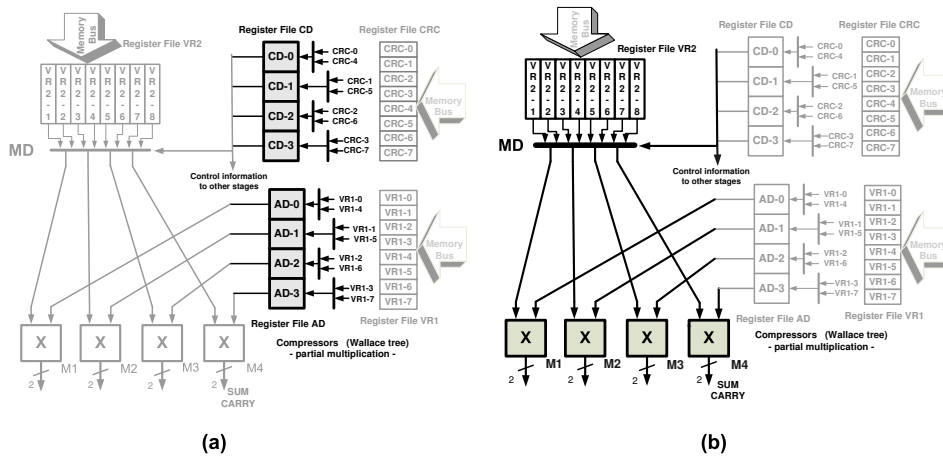


Figure 4.7: Multiplication reduction trees

Stage 3) Reduction Trees (Multiple-Addition): The third pipeline stage is in charge of compute the multiple operand addition, fundamental for processing in parallel 4 data inputs of dense or sparse matrices. This stage is composed by four parallel reduction trees, each one intended to add several operands. Figure 4.8 depicts four main compressors: (6:2), (8:2), (10:2) and (12:2) used to reduce the incoming data as explained in the followings.

⁴Multiplexers are represented by thick lines in the figure's.

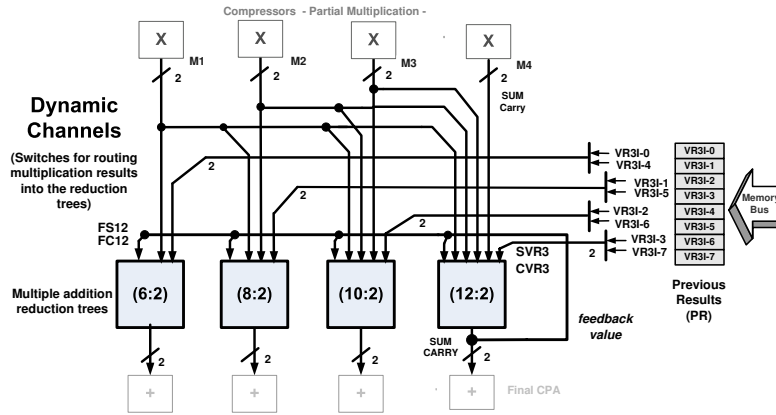


Figure 4.8: Reduction trees (multiple-addition)

The multiple-operand addition hardware compressors are designed to perform the following operations:

- The (6:2) compressor is used to carry out the partial addition of one multiply reduction tree outcome (M1), the partial result of the computing row (12:2) result (*feedback value*), and the Previous Result (PR) stored in VR3I;
- A reduction tree (8:2) receives two partial product outcomes of M1 and M2, the *feedback value* and the corresponding VR3I input;
- For adding three partial products (M1, M2 and M3), a *feedback value* and the VR3I data, it is used a (10:2) reduction tree;
- The (12:2) reduction tree is expended to compute the partial addition of four incoming partial products M1, M2, M3 and M4; the VR3I value and the *feedback value*.

Resource Allocation Mechanism: The allocation of the multiplication results into four parallel counters (6:2), (8:2), (10:2), and (12:2) is controlled by the EOR flags of the incoming set ($s = 4$). Table 4.2 shows 4 variables: W, X, Y and Z in the first column. These variables corresponds to the EOR flags associated with the 4 nonzero A_m elements being processed, where W is the first entry and Z the last entry of the input set. All 16 possible combinations of the 4 EOR flags are considered. Based on the pattern of the EOR flags, a

control allocation algorithm is presented. The bits presented in columns two to five in Table 4.2, are used to control the issue of the multiplication-results into the group of proposed multiple addition reduction trees. The distribution is achieved with the use of a group of multiplexers denominated here “Dynamic Channels”. This information is a fundamental key for the entire unit operation.

Table 4.2: Resources allocation control bits.

EOR W X Y Z	(6:2)	(8:2)	(10:2)	(12:2)	Compressor used.	\bar{Z}
0 0 0 0	000000	00000000	0000000000	FF0011111111	(12:2)	1
0 0 0 1	000000	00000000	0000000000	FF1111111111	(12:2)	0
0 0 1 0	000000	00000000	FF11111111	000000000011	(10:2),(12:2)	1
0 0 1 1	000000	00000000	FF11111111	001100000011	(10:2),(12:2)	0
0 1 0 0	000000	FF111111	0000000000	000000001111	(8:2),(12:2)	1
0 1 0 1	000000	FF111111	0000000000	001100001111	(8:2),(12:2)	0
0 1 1 0	000000	FF111111	0011000011	000000000011	(8:2),(10:2),(12:2)	1
0 1 1 1	000000	FF111111	0011000011	001100000011	(8:2),(10:2),(12:2)	0
1 0 0 0	FF1111	00000000	0000000000	000000111111	(6:2),(12:2)	1
1 0 0 1	FF1111	00000000	0000000000	001100111111	(6:2),(12:2)	0
1 0 1 0	FF1111	00000000	0011001111	000000000011	(6:2),(10:2),(12:2)	1
1 0 1 1	FF1111	00000000	0011001111	001100000011	(6:2),(10:2),(12:2)	0
1 1 0 0	FF1111	00111100	0000000000	000000001111	(6:2),(8:2),(12:2)	1
1 1 0 1	FF1111	00111100	0000000000	001100001111	(6:2),(8:2),(12:2)	0
1 1 1 0	FF1111	00110011	0011000011	000000000011	(6:2),(8:2),(10:2),(12:2)	1
1 1 1 1	FF1111	00110011	0011000011	001100000011	(6:2),(8:2),(10:2),(12:2)	0

The final strategy to allocate the multiplication results into the multiple additions hardware is presented in the sixth column of Table 4.2, labeled “Compressor used”. This strategy is tuned to optimize the control complexity, reducing the control logic from 16 to only 2 main states. Reduction is made upon consideration of the following 4 rules:

1. The processing of data associated with EOR flags with even terms such as: $WXYZ = 0, 2, 4, 6, 8, 10, 12$ and 14 (always with $Z = 0$) means that currently a row is still computing. Thus, it will need to add its partial result, held as the *feedback value*, to the rest of row values derived from the next four inputs. This partial computing is performed by the (12:2) compressor. Therefore, the *feedback value* will always come from this multiple operands addition tree (see Figure 4.8). By this rule the feedback control is then simplified because we can rule out the rest of the resources as possible sources of a feedback data;
2. Allocation of resources follows the principle of using the biggest compressor when possible, taking always into account rule 1;

3. When possible, the currently processed nonzero elements with flags W, X, Y and Z are distributed to compressors (6:2), (8:2), (10:2) and (12:2) respectively;
4. EOR flags enable the use of VR3I values for proper accumulation of partial or final values.

The nomenclature used to represent allocation control bits in Table 4.2, uses a ‘1’ to enable the dynamic connecting channels⁵ of data, and a ‘0’ to disable the issuing of data into the multiple addition reduction trees. We enumerate the bits in the control words from MSB to LSB; being bit 0 the MSB. Thus, for any compressor, from left to right, we can determine the following meaning and behavior of the bits:

1. Bits 0 and 1 are used to control the incoming result from the (12:2) compression tree outcome. Using these bits we can compute n DMVM sets of 4 data with a supporting hardware of $s = 4$. Some entries have an ‘F’ symbol. We use symbol ‘F’ to show that the dynamic connecting channels of data will be either enabled or disabled to issue data into the reduction trees. This occurrence will depend on the Z value of the previous computed set. Therefore, when processing, e.g., the small matrix of section S^2 , see Figure 4.1, partial result feedback is needed to obtain the row’s final result, since we need to add the result of the first quadruple 1, 2, 3, 4 to quadruple 9, 2, 7, 9 values. Consequently, equations (4.4) and (4.5) represent the possible feedback data $FS12_i$ and $FC12_i$ (see Figure 4.8) that come from the (12:2) compressor:

$$FS12_i = S12_i \cdot \bar{Z} \quad \forall 0 \leq i \leq 31 \quad (4.4)$$

$$FC12_i = C12_i \cdot \bar{Z} \quad \forall 0 \leq i \leq 31 \quad (4.5)$$

where $S12_i$ and $C12_i$ are the SUM and CARRY outcomes of (12:2) compressor tree respectively, and “ \cdot ” represents the logical AND operation. Furthermore, these bits also control the feedback of a partial result when processing a sparse-matrix-vector-multiply operation.

2. Bits 2 and 3 are used to control the issuing of VR3I values, SUM ($VR3SI_i$) and CARRY ($VR3CI_i$). This operation is performed when

⁵Regards the path followed by a data coming from the multiply reduction tree outcome to the multiply-addition reduction tree.

EOR flag is ‘1’ for any of the 4 inputs. Logical equations (4.6) and (4.7) describe the control of this operation (see Figure 4.8 for these signals).

$$SVR3_i = VR3SI_i \cdot EOR \quad \forall 0 \leq i \leq 31 \quad (4.6)$$

$$CVR3_i = VR3CI_i \cdot EOR \quad \forall 0 \leq i \leq 31 \quad (4.7)$$

where $SVR3_i$ and $CVR3_i$ represents the possible SUM and CARRY values added into the compressor trees. For example, Figure 4.1 shows that to compute row 24 in section A^3 we need to add the value ‘1’ to the previous result of section A^2 , completing thus (as suggested by equation (4.3)) the computation of the row.

3. Finally, from left to right, starting from bit 4 the remainder of the bits controls the multiplication compressor outcomes’ delivery. The next 2, 4, 6 and 8 bits enable or disable inputs to the (6:2), (8:2), (10:2) and (12:2) compression trees respectively, basically they control the issuing of M1, M2, M3 and M4 results. The control of each multiplication reduction tree output (SUM and CARRY) is described by the following equations:

$$S = SM_i \cdot SE \quad \forall 0 \leq i \leq 31 \quad (4.8)$$

$$C = CM_i \cdot CE \quad \forall 0 \leq i \leq 31 \quad (4.9)$$

where SM_i represents the SUM and CM_i the CARRY outputs from the partial multiplication unit, e.g. M1, and Sum Enable (SE) and Carry enable (CE) are used in this equation for representing each pair of control bits presented in Table 4.2 from position bit 4. It is important to notice that a set of multiplexers is necessary to chose the correct state being processed. The input to those multiplexers comes from equations (4.4) to (4.9).

Stage 4) Final Addition and Result Update: The fourth stage reduces and finalizes computing of the multiple-addition trees’ results. The 4 final adders outcomes are stored according to the column flags, and 4 de-multiplexers accomplish the task as schematized in Figure 4.9. The main memory should be updated from registers file VR3O.

4.2.2 Sparse Matrix-Vector Multiply Example

In this subsection, we present an example of sparse matrix multiplication for all targeted formats with nonzero elements, a column pointer as well as an EOR

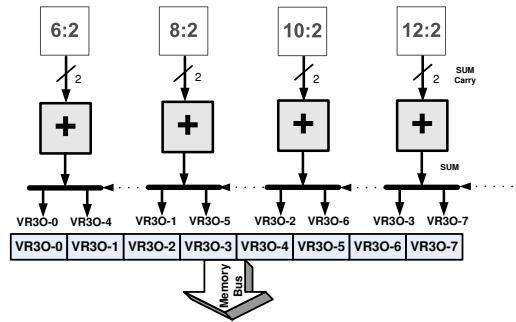


Figure 4.9: Final addition

flag as proposed in Section 4.1.1. The pipeline structure described above computes in parallel 4 inputs at the time. In Figure 4.10 (a) the NZE (2, 8, 2 and 5) has associated the EOR flags (WXYZ) equal to “0110”. Using the resource allocation strategy presented in Table 4.2, the multiplication results, produced by the multiply reduction trees are distributed in this case into the following multiple operand addition resources: (8:2), (10:2) and (12:2) for their addition.

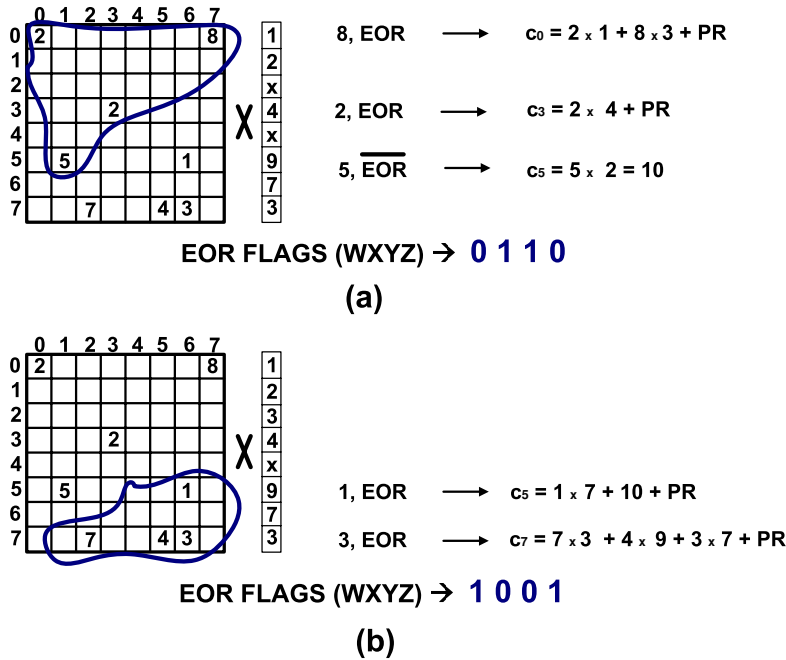


Figure 4.10: Sparse by dense vector multiplication for s=4: a) First processing step, b) Second processing step.

Partial result c_0 is obtained from the (8:2) compressor, that adds two multiplication results plus a previous result (PR) ⁶. c_3 is equal to the addition of one multiplication result and PR, this addition is carried out into the (10:2) compressor; while c_5 has only one value. The last NZE from the first group considered in Figure 4.10(a) (i.e., number “5”) belongs to a row that is still being processing due to EOR = 0. The value 10 ($=5 \times 2$), it is passed trough the (12:2) compressor padded with zeros and remains in the pipeline for the computing of the next 4 inputs (see the internal feedback from the output of (12:2) compressor in Figure 4.8). Figure 4.10(b) depicts the processing of the next 4 inputs with EOR = “1001”. c_5 is now finally computed into the (6:2) compressor, which is equal to the previous value computed and retained (that always comes from the (12:2) compressor), plus the current multiplication result and the PR. Finally c_7 is the result of the addition of 4 values, 3 correspond to the multiplication results and the last one to a PR value, this operation is carried out in (12:2) compressor.

4.2.3 Reconfigurable Optimizations:

High parallel algorithms can use several reconfigurable units for speeding up the processing of dense-matrix-vector-multiply operation. Those algorithms can schedule in each unit an inner loop described by equation (4.3). In our case to achieve scalability, in each unit, instead of using multiplexers to route the data, trees connectivity are reconfigured, improving in this way the processing delay and limiting the hardware used. A group of simplified units for dense matrix processing is presented in Figure 4.11

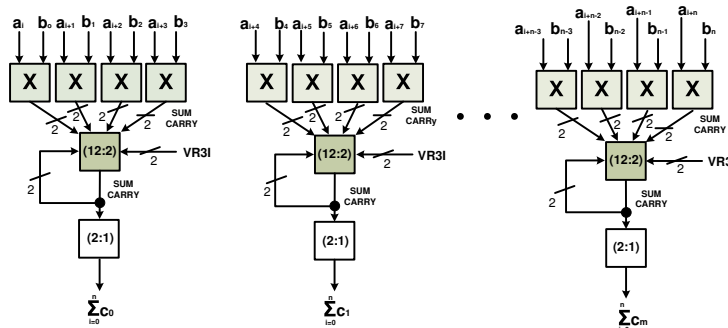


Figure 4.11: Scalability on dense matrix processing

⁶PR: comes from a previous sub-matrix computing

Performance improvement is achieved and in the same time the size of the entire unit is reduced with this approach. This is done with the use of the reconfigurable characteristics offered by FPGAs, modifying partially the pipeline unit, leaving the first stage as it is, using a simple MUX instead of MD in the second stage, leaving only the (12:2) multiple-operand adder in the third stage, and finally, using only one final (2:1) compressor instead of the four used in the original unit proposed. Assuming a polymorphic processor scenario, as presented in [16], the reconfigurable coprocessor will behave as follows:

1. A group of necessary (3:2)counters can be implemented at fabrication time. Using these resources, we can set up in advance the main blocks like: the partial multipliers, the multiple-operand addition units like the (12:2), as well as, the final (2:1) adders;
2. A partial reconfiguration of the structure can be implemented on demand, setting up the dynamic channels in order to support only the desired operation. Furthermore, the same basic hardware can be used to support also other operations, which are based on addition and multiple addition related operations like the ones discussed in the previous Chapter 3.

The two above presented points (1) and (2) are suitable to the reconfigurable processor platforms with partial reconfiguration capabilities.

4.3 Experimental Results

Our proposal is intended to cover various formats, i.e. sub words in fixed point notations, and incorporate both: sparse and dense computation into a single unit; opposed to other approaches that are intended for scientific computations and employ either sparse or dense matrices in floating point formats. We have presented a general design intended to be used in ASICs as well as in reconfigurable technology. The proposed unit was described using VHDL, synthesized with the ISE 6.1i Xilinx environment [78], for a xc2vp100-6 FPGA device. The synthesis results indicates that our design utilizes 4255 slices, 1505 FFs and 6472 LUTs for the entire unit. Additionally, we synthesize each particular pipeline stage separately in order to find the contribution of each stage into the overall results. We did this in respect of area and delay; those results are

depicted in Table 4.3 ⁷.

Table 4.3: Matrix-vector multiply/add unit.

(Pipelines stages: time delay - hardware use)

Xilinx XC2VP100	Partial Multiply	Multiple-Addition.	Final Add
Operand Width	16	32	32
Pipeline Stages	1	1	1
# of Slices	1656	1595	332
LUT	3192	3174	640
Area utilization %	37	35	7
Latency (ns)	8.3	6.0	6.3

From the above table, we can conclude that the partial multiply stage is the bottleneck that limits frequency of the unit's operation. We notice that the partial multiplier unit without the final addition logic is equivalent in terms of time response to the built-in multipliers on the FPGA. Nevertheless, this stage, like others, can be pipelined in order to improve the unit's throughput. Concerning the hardware used and presented in Table 4.3, the second stage consumes 37% of the designed unit implementing the four multiply reduction trees and the routing logic. The third stage requires similar hardware to the previous one, and 35% of the unit is used by the four multiple-addition trees with their corresponding routing logic. The final 4 adders utilize 7% of the hardware used. Those three stages use 79% of the total hardware. Regarding the multiplexers used to route data into the second stage depicted in Table 4.4, we should mention that the 1.3 ns introduced as an extra delay become a 16 % of the total stage delay.

Table 4.4: Routing multiplexers - Second Stage.

(Routing hardware: - time delay and hardware use)

Xilinx XC2VP100	M-CD	M-AD	MD
Operand Width	8	16	16
# of Slices	18	37	128
LUT	32	64	256
Latency (ns)	0.4	0.4	1.3

⁷For first stage see Table 4.6. Independent synthesized pipeline units use more hardware when compared to the whole unit, due to synthesis introduces resource sharing.

In reference to the area used to route the output of the 4 partial-multipliers, we found a maximum of 1.8 ns of extra delay as shown in Table 4.5. The extra delay represents 30% of the total stage delay. This information suggests a bigger use of resources compared with the previous stage. This occurs because of the operands size is increased due a partial multiply outcomes, it can be noted that we use SUM and CARRY representation in our pipeline.

Table 4.5: Allocation hardware -Third stage.

(Dynamic channel hardware: time delay and hardware use)

Xilinx XC2VP100	G-6:2	G-8:2	G-10:2	G-12:2
Operand Width	32	32	32	32
# of Slices	32	147	186	214
LUT	192	256	324	388
Latency (ns)	0.4	0.4	1.6	1.8

The remainder of used area, is employed to built the register files and pipeline registers as illustrated in Table 4.6.

Table 4.6: Matrix-vector multiplication unit.

(File registers: time delay - hardware use)

Xilinx XC2VP100	VR1/VR2	CR/Control	AD/CD	VR31/VR30	R-Pipeline
Operand Width	16	16/8	8	32/32	64
# of Slices	72/72	40/40	36/20	144/144	288
Flip-Flips	128/128	64/64	64/32	256/256	512
Area utilization %	3.2	2.8	2.2	6.4	6.4
Latency (ns)	1.6	1.6	1.6	1.6	1.6

Control: refers to the control register that holds in the pipeline the Column and EOR information.

Additionally, we also synthesize the reduced unit presented earlier. Such reduced unit use 2507 slices and 3904 LUTs, representing 60% of the original unit. Also the frequency operation increments to 123 MHz. It is estimated that a 1/2 of the (3:2)counters used to construct the unit are shared by the scaled one. Therefore, the proposed unit can be considered as valid (viable) candidate block, suitable to work in a reconfigurable collapsed arithmetic unit.

4.4 Conclusions

We have presented a novel fixed point integer matrix-vector multiplication unit suitable to work with sparse and dense matrices found in several media formats. The acceleration of the proposed unit is achieved using a new concurrent and scalable hardware capable of processing 4 matrix entries per cycle, carrying out 4 multiplications and up to 12 additions at 120 MHz using 9% of the resources in a VIRTEX II PRO xc2vp100-6 FPGA device. We further presented pre-processing of different formats in order to convert the CRS and HiSM formats to an equivalent BBCS compression format, without incurring significant area and time overhead. In all of the above representations we can extract the information of the last nonzero element of a row, and analyzing the data, we introduced an allocating algorithm which distributes efficiently the incoming data into parallel support hardware. A total of 11 units can be integrated in the same FPGA chip, achieving a performance of 21 GOPS. A 40% of hardware reduction is achieved using the reconfiguration capabilities of FPGA devices when operated as a scaled and modified unit.

Chapter 5

Arithmetic Unit for Universal Addition

Various applications, e.g. commercial and financial electronic transactions [79], internet [80] and industrial control [81] require precise arithmetic for different data representation formats. Binary arithmetic is not capable of expressing exactly fractional numbers like, 0.2, and therefore, Binary Decimal Arithmetic emerges as a possible solution [33] to avoid binary approximations [82, 83]. When performing decimal operations on traditional binary based hardware, excessive delays are introduced due to the software emulations, conversions and corrections, typically 100 to 1000 times slower [84]. Therefore, flexible hardware solutions for both decimal and binary processing are considered in this chapter. We assume universal units similar to those presented in [42] that are capable of performing various related operations sharing the same hardware. More specifically, the main contributions of this chapter are:

- Twelve related operations were collapsed into a single Universal Adder (UA) hardware unit using Universal-Notationⁱ. The proposed structure uses the *effective* addition/subtraction approach similar to [54] for both, binary and BCD notations.
- High hardware utilization: approximately 40% of the hardware resources implementing the 12 different operations are reused. This makes

ⁱIn the context of this chapter, we assume operands and results to be in unsigned, sign-magnitude or complement (two's complement for binary and ten's complement for decimal) notations.

the unit a good candidate for implementation on partially reconfigurable platforms such as [16].

The remainder of this chapter is organized as follows. Section 5.1 outlines the background and presents the related work. Section 5.2 exhibits with details the Reconfigurable Universal Adder design. Section 5.3 presents the experimental results and introduces the analysis in terms of used area and delay. Finally, Section 5.4 concludes the Chapter.

5.1 Background

First, we note that the mathematical notations, used in this chapter deviate from the one introduced for Chapters 2, 3 and 4. The reason are that we do not consider matrix operations in this chapter and that the referenced related work was originally presented in different notation. Shortly in this chapter we consider only one dimensional vectors and individual bits, therefore we use appropriate notations for concise representations.

Decimal digits are usually converted to binary representation as follows: the 8421 BCD code uses only the first ten combinations of a 4-bit binary code to represent each decimal digit. The remaining encodings (1010 to 1111) corresponding to decimal (10 to 15) are left unused when decimal computing is considered. Assuming addition of two decimal numbers $N1$ and $N2$ their SUM can exceed 1001_2 ($SUM > 9$). In such a case, correcting addition using 0110_2 (6) is required, so the complete operation becomes $SUM = N1 + N2 + 0110_2$. Please note that in case of subtraction $N2$ can be the complement representation of the original operand. In other words, decimal subtraction introduces additional processing. Recall that a complement (CN) of an n -bit number N is computed by: $CN = r^n - N$; where r is the radix of the number ($r = 2$ for binary, and $r = 10$ for decimal). Thus, the ten's complement of an x -digit number N , is expressed by $CN_N = 10^x - N$. When we consider only a single digit D (as in the case with BCD), the computation is simplified to $CN_D = 10 - D$. Nevertheless, BCD encoding do not include a code for 10 and for this reason a nine's complement representation is used. In this case the computation becomes: $CN_D = 10 - D = 9 - D + 1$. Figure 5.1 depicts the steps involved in a BCD subtraction operation including nine's complement and the correcting addition with 6.

	<u>Decimal Subtraction</u>	<u>Ten's Complement</u>	<u>BCD addition</u>
N1	1 5 7 8	9 9 9 9	1 5 7 8
N2	$\begin{array}{r} 1578 \\ - 545 \\ \hline 1033 \end{array}$	$\begin{array}{r} 9999 \\ - 545 \\ \hline 9454 \\ + \quad 1 \\ \hline 9455 \end{array}$	$\begin{array}{r} 1578 \\ + 9455 \\ \hline \overset{1}{A} \overset{9}{9} \overset{C}{C} \overset{1}{D} \\ + 6666 \\ \hline 1033 \end{array}$

Figure 5.1: Decimal subtraction

Some helpful techniques used to compute a nine's complement of a digit are [85]:

Pre-complement In this approach the subtrahend is one-complemented, a 1010_2 value is added for correction, and finally the generated carries are discarded.

Post-complement In this case, a binary 0110_2 value is added to the subtrahend operand, the result is one-complemented and the generated carries are used.

5.1.1 Related Work

Many researchers addressed the problem of decimal arithmetic in the past. Early solutions proposed customized decimal adders, like Schmookler et al. [86] and Adiletta et al. [87]. Combined Binary and BCD adders were designed by Levine et al. [88] and Anderson [89], while true decimal sign-magnitude adder/subtractor was presented by Grupe [90]. The latter used 3-binary-adders along with additional logic, achieving similar results as presented in this chapter. An area efficient sign-magnitude adder was developed by Hwang [1]. In his approach two additional conversions are introduced before and after the binary addition. This is somehow similar to other proposals, e.g. [90]. The novelty in Hwang's proposal comes with the separation of the binary and the decimal results, using a multiplexer to select the correct output as depicted in Figure 5.2.

Flora [91] presents an adder that process concurrently two different results, one assuming the presence of an input carry and the other assuming no carry in available. This is following the principle of carry select adders [48], an approach actually used in several state of the art units. To cope with the area overhead, another compact design employing a single adder was introduced by Fischer et al. [2] it is depicted in Figure 5.3. This unit uses the control signals

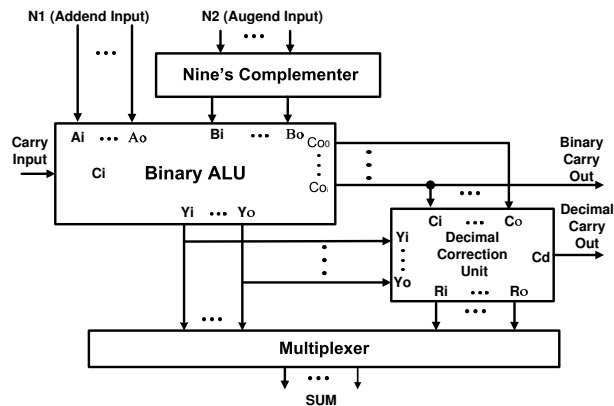


Figure 5.2: Hwang's proposal [1]

$F1$ and $F2$ to encode (“edit”) the incoming operands; signal $F3$ is used to control the encoding of the binary adder output in order to achieve the desired functionality. Nevertheless, the authors do not provide the automatic mechanism to identify the possible scenarios for using the appropriate $F1$, $F2$ and $F3$ values. Furthermore, the encoder proposed in this work at the input and output of the binary adder unit does not contemplate the case when subtrahend is greater than the minuend. The critical path is determined by the binary adder and by the two encoders.

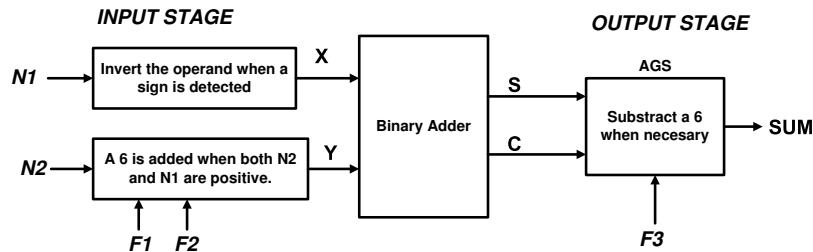


Figure 5.3: Fischer's proposal [2]

During the last decade various binary and BCD adder/subtractor units for the IBM S/390, G4, G5 and G6 microprocessors [92] have been developed. Following this line, the eServer z900 processor [93] includes a combined binary/decimal arithmetic unit working with binary-coded decimal numbers in sign-magnitude representation. The z900 unit was designed to operate in a

single cycle, nevertheless a correction of the results is required in some of the cases [94]. For example, when an effective subtraction operation is performed, they need to calculate the complement value of the result (for binary and decimal results), hence increasing the latency. The approach used to construct this decimal unit is based on the work presented in [95], which is shown in Figure 5.4. In this approach, the latency of other approaches is reduced because of the carry select organization [48] used to accelerate the computing. Two adder structures compute in parallel the addition/subtraction operation. Pre-Sum-0 assumes a Carry-in = '0'; while Pre-Sum-1 assumes a Carry-in = '1'. A final multiplexer base on the true computing of the Digit Carry Network (see Figure 5.4) chooses the appropriate SUM computed value. Recently, a similar approach was presented by Haller et al. [96]. In this work, the carry chain of the Digit Carry Network is optimized resulting in a slight delay improvement with an increased area of the unit.

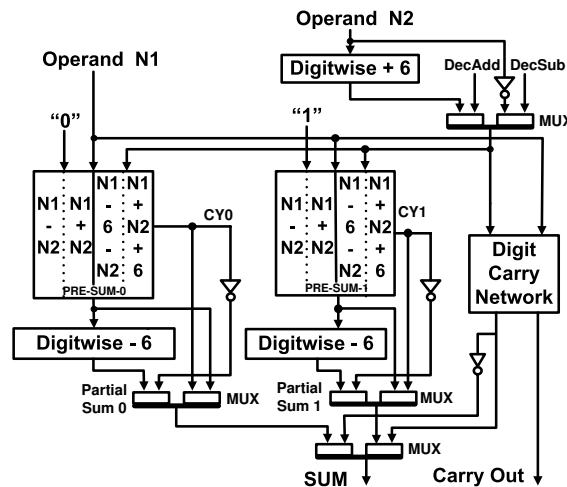


Figure 5.4: Haller's proposal (z900)

The following Table 5.1, summarizes the fundamental characteristics of the adders discussed above. The subset of the proposals aiming at signed arithmetic do not produce the correct result and need an additional complement operation in the cases when the subtrahend is greater than the minuend, this means that those support hardware will require an additional cycle to produce the correct result. As will be shown in the text to follow our approach does not have this disadvantage, requiring only one cycle to produce the correct result (see subsection 5.2.2 for an example).

Table 5.1: Adders - data representation

Adder	BCD		Binary	
	Addition	Subtraction	Addition	Subtraction
Schmookler [86]	U	-	-	-
Addileta [87]	U	-	-	-
Levine [88]	U	-	U	-
Anderson [89]	U	-	U	-
Grupe [90]	S	S	S	S
Hwang [1]	S	ten's	S	two's
Flora [91]	S	ten's	S	two's
Fischer [2]	S	ten's	S	two's
z900 [94]	S	ten's	S	two's
Haller2006 [96]	S	ten's	S	two's

U: unsigned; S: sign-magnitude. Two's and ten's: for complement representation.

5.2 Reconfigurable Universal Adder

The main goal of our work was to create an adder capable of carrying out decimal operations using effective addition/subtraction scheme, overcoming the limitations of the adders presented in the related work section. To achieve this we used the S/370 sign-magnitude binary adder presented in [54] as a base. We extend its functionality to perform decimal addition/subtraction operations along with the original binary ones. Next, we briefly introduce the original binary adder and focus on our extensions. In the next subsection we will discuss the specific decimal functionalities of our design.

Let us assume $N1$ and $N2$ being two n -bit sign-magnitude numbers, such that $N1 = [N1_{n-1}N1_{n-2}...N1_0]$ and $N2 = [N2_{n-1}N2_{n-2}...N2_0]$, with $N1_{n-1}$ and $N2_{n-1}$ used as sign bits of binary or BCD representations. We propose to modify the sign bits to cope with the cases of unsigned and complement numbers as will be explained later. The two modified sign bits are computed as follows:

$$N1_S = N1_{n-1} \cdot Type \quad (5.1)$$

$$N2_S = N2_{n-1} \cdot Type \quad (5.2)$$

where $Type = 0$ signal, masks both sign bits $N1_{n-1}$ and $N2_{n-1}$ in the case of unsigned and complement representations.

Effective Addition/Subtraction Operation: Assuming *absolute values*ⁱⁱ such as in the S/370 sign-magnitude binary adder, equation (5.3) is used to determine the exact operation, e.g. effective addition either subtraction. The logic one value of the signal Effective Addition ($EAdd$) indicates an effective addition operation, logic zero indicates that an effective subtraction is carried out.

$$\begin{aligned} EAdd &= (\overline{N1_S} \cdot \overline{N2_S} \cdot \overline{Add}) | (\overline{N1_S} \cdot N2_S \cdot Add) | \\ &\quad (N1_S \cdot \overline{N2_S} \cdot Add) | (N1_S \cdot N2_S \cdot \overline{Add}) \\ &= (N1_S \oplus N2_S \oplus \overline{Add}) \end{aligned} \quad (5.3)$$

where input Add indicates the desired operation ($Add=0$:add, $Add=1$:subtract). Here in this chapter “|” is used to denote logical *OR* operation and “+” for addition.

Notice that, when $Type = 0$, $EAdd$ become equal to \overline{Add} . This is essential for the correct processing of the unsigned and complement (ten’s and two’s) operations. In addition, we should use in some cases the $N1_{n-1}$ and $N2_{n-1}$ values for the addition (SUM). How, those values participate into the above operation for all possible cases are stated in Table 5.2.

Table 5.2: Adder setting up considerations

Representation	$EAdd$	SUM	$Type$
Signed-Magnitude	$N1_{n-1}, N2_{n-1}$	$N1_{n-1} = N2_{n-1} = 0$	1
Unsigned.	$N1_{n-1} = N2_{n-1} = 0$	$N1_{n-1} = N2_{n-1} = 0$	0
Complement (Ten’s & Two’s)	$N1_{n-1} = N2_{n-1} = 0$	$N1_{n-1}, N2_{n-1}$	0

The binary sign-magnitude addition is performed using the absolute values of the input addends. Then the following equation (5.4) establishes this operation:

$$\sum = |N1| + |N2|^* \quad (5.4)$$

where $|N2|^*$ is equal to $|N2|$ for effective addition and $|N2|^*$ equals to $\overline{|N2|}$ in case of effective subtractionⁱⁱⁱ. In order to generate a correct sign-magnitude

ⁱⁱ Absolute values assume only the magnitude of a signed magnitude number.

ⁱⁱⁱ $EAdd$ signal is used to control the complement operation of $N2$.

result, an additional correction step is used. The final magnitude result becomes:

$$|SUM| = \sum_k \oplus \Delta \quad \forall \quad 0 < k < n - 2 \quad (5.5)$$

where \oplus is the exclusive OR operator and the Δ is:

$$\Delta = \overline{Co} \cdot \overline{EAdd} \quad (5.6)$$

with carry output (Co) equal to:

$$\begin{aligned} Co &= G_0^{n-2} [|N1|, |N2|^*] \\ &= G_0 (P_0 \cdot G_1 | \dots (P_0 \cdot P_1 \dots \cdot P_{i-1} \cdot G_i \\ &\quad | \dots (P_0 \cdot P_1 \dots \cdot P_{i-3} \cdot G_{n-2})) \end{aligned} \quad (5.7)$$

where G_i and P_i are the *generate* and *propagate* signals of the single bit adders. Finally, the sign bit of the result is updated as shown in equation 5.8 (see [54] for details):

$$SUM_{n-1} = [N1_{n-1} \oplus (\overline{Co} \cdot \overline{EAdd})] \cdot \overline{(SUM \equiv 0)} \quad (5.8)$$

with $SUM \equiv 0$ represents SUM equal to zero.

All of the different cases of the original adder are presented in the table shown in Figure 5.5.

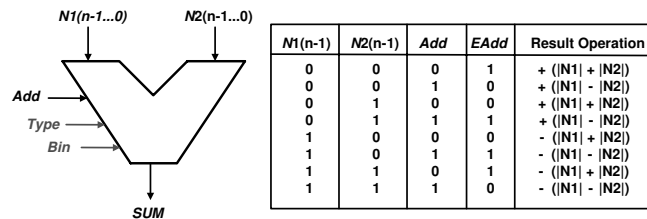


Figure 5.5: Sign magnitude adder

For decimal operation we reuse the $EAdd$ and Add signals and introduce the two additional control signals (Bin and $Type$). $Bin = 1$ causes a binary operation, otherwise a BCD operation is performed. The $Type$ signal has the functionality as previously described.

The digit carry logic (DC) [94] signals for decimal operations are obtained as follows:

$$DC = X \mid Y \cdot CI \quad (5.10)$$

where

$$\begin{aligned} X &= G_3 \mid P_3 \cdot P_2 \mid P_3 \cdot P_1 \mid G_2 \cdot P_1 \\ Y &= P_3 \mid G_3 \mid P_1 \cdot G_1 \\ CI &= G_1 \mid P_1 \cdot C_{in} \end{aligned}$$

All necessary correction cases for decimal data arithmetic, proposed in this work, are summarized in Table 5.3. Our coder is slightly more complex than previous proposals, it contemplates several correction terms and enables efficient addition-subtraction for our universal adder. Note, that in the post-complement operation case, two necessary correction operations are sometimes required, both using 0110_2 value (one on the subtrahend value and one on the correction of addition result). In our proposal, in contrast to all previous proposal, for example a single addition with the correction term 1100_2 is used selectively. The selective utilization of correction terms allows effective addition/subtraction operations.

Table 5.3: Decimal digit correction terms

Operation	DC		\overline{DC}	
	$N1 > N2$	$N2 \geq N1$	$N1 > N2$	$N2 \geq N1$
Addition	0110_2	0110_2	0000_2	0000_2
Subtraction	0110_2	1100_2	0000_2	0110_2

Please note that when a binary operation is performed $Bin = 1$ the decimal correction term is not needed. A value of 0000_2 is used for any binary operation. The universal adder is set up with the aforementioned logic, a one-dimensional (3:2)counter array, a carry-propagate-adder and a set of XOR gates. The final organization is depicted in Figure 5.7.

Note that the input $N2^*$ for computing the digit carry logic is equal to $\overline{N2} + 10$ when processing decimal subtraction otherwise is equal to $N2$. The multiplexer signal control for decimal subtraction or any addition (DS) is computed by:

$$DS = \overline{Add} \cdot \overline{Bin} \quad (5.12)$$

The final complement operation is controlled by equation (5.13) which is a

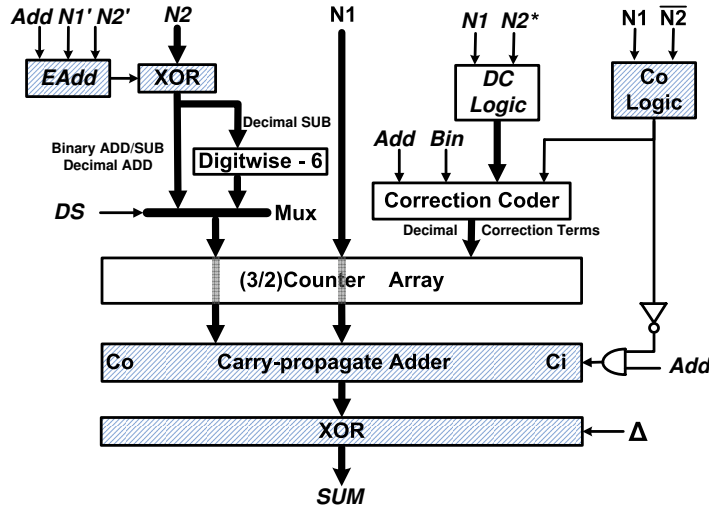


Figure 5.7: Universal adder micro-architecture [3]

modified version of equation (5.6):

$$\Delta = (\overline{C_o} \cdot \overline{EAdd}) | (C_o \cdot \overline{Add} \cdot \overline{Bin}) \quad (5.13)$$

The rationale behind the design of the universal adder is based on the following:

- $EAdd$ signal controls one's complement operation of the subtrahend in binary and decimal operations.
- The Δ logic (see equation (5.13)) corrects the final result in binary and decimal operations when necessary.
- We need a (3:2)counter row to selectively add the necessary correction terms (correction coder) as indicated by Table 5.3.
- The hardware reuse is fundamental when a partial reconfigurable hardware unit is designed for reconfigurable processor scenarios.

When our design is mapped on partially reconfigurable hardware platforms, the modules shown in dark in Figure 5.7, such as: the carry-propagate-adder, the $EAdd$ logic, the two XOR logic blocks (at the input and output of the unit) and the Co logic; can be reused in both modes. These “permanent” blocks can be configured at the beginning when the processor is set up. In order to provide the BCD functionality the remaining (shown as not colored in Figure 5.7)

modules can be configured on demand. Such dynamic partial reconfiguration will diminish the penalty due to reconfiguration latencies. Our estimation [97] indicates that the “permanent” blocks account for approximately 40 % of the total hardware resources implementing the operations when targeting Xilinx Virtex4 devices.

Finally, the fundamental decimal addition and subtraction operations of our proposal can be summarized with the following equations^{iv} :

Decimal Addition

$$Sum = N1 + N2 + 0110_2 \vee DC = 1$$

$$Sum = N1 + N2 + 0000_2 \vee DC = 0$$

Decimal Subtraction

$$Sum = N1 + DW(\overline{N2}) + 0110_2 \vee DC = 1 \wedge Co = 0$$

$$Sum = N1 + DW(\overline{N2}) + 0000_2 \vee DC = 0 \wedge Co = 0$$

$$Sum = N1 + DW(\overline{N2}) + 1100_2 \vee DC = 1 \wedge Co = 1$$

$$Sum = N1 + DW(\overline{N2}) + 0110_2 \vee DC = 0 \wedge Co = 1$$

5.2.2 Decimal Subtraction Example

Figure 5.8 shows the decimal subtraction of $N1 = 45_{10} = "0100\ 0101"_{BCD}$ and $N2 = 78_{10} = "0111\ 1000"_{BCD}$.

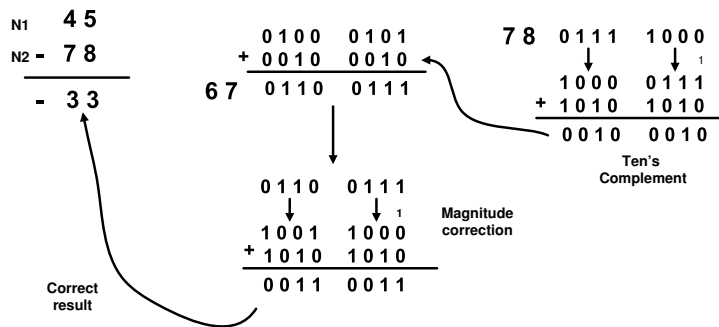


Figure 5.8: Decimal subtraction: double addition scheme

^{iv}Please note that the decimal *Sum* value presented in those equations requires the one complemented operation as was stated in equation (5.5). Where the Δ control signal (5.13) is used to obtain the final *SUM* result

The example depicted in Figure 5.8, for a subtrahend greater than the minuend ($78 > 45$), evidences the necessary additional ten's complement operation. Previous proposals (see background section), do not operate with post-complement operation. Those solutions provide an incorrect 67_{10} value instead of the 33_{10} final result. On the other hand, Figure 5.9 presents an example of the computing algorithm carried out by our arithmetic accelerator. In this example we have: $N2 > N1$ and the digit carry (DC) equal to one. Therefore, a correction term for each digit is required according to Table 5.3. Additionally, $\Delta = 0$ points out that the result does not require the one's complement operation correction. Thus, the result is simplified to add three numbers. Our solution uses a structure of a (3:2)counter and a final carry propagation unit (see for implementation details Figure 5.7).

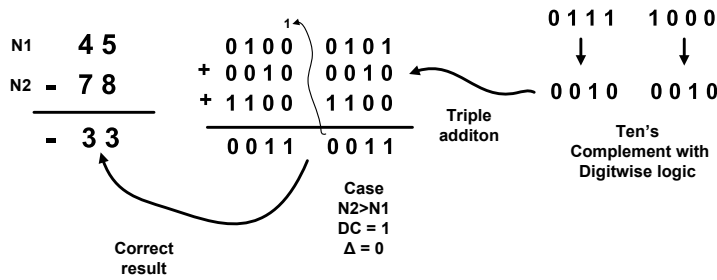


Figure 5.9: Decimal subtraction: collapse approach

5.3 Experimental Results Analysis

The proposed Universal Adder was implemented using VHDL, synthesized, functionally tested, and evaluated using the ISE 8.1i Xilinx design tools [57] targeting 4vfx60ff672-12 VIRTEX 4 FPGA device. Furthermore, the designs proposed by Fischer [2], Hwang [1], Busaba [94] and Haller [96] were also implemented and synthesized using the same methodology. Table 5.4, summarizes the latency and hardware utilization results for all of the considered designs.

Table 5.4: Latency&area - Adder comparison

32-bit wide units	Latency (ns)			Resources used	
	Logic	Wire	Total	Slices	LUTs
Our proposal	7.3	4.7	12.1	256	495
Fischer [2]	5.5	4.8	10.3	123	233
Hwang [1]	5.9	4.6	10.5	82	158
Busaba [94]	5.3	5.6	10.9	242	466
Haller [96]	5.5	4.5	10.0	305	584

Please note that all of the units proposed before require an additional cycle to produce the correct result in cases when $N2 > N1$ for the BCD subtraction operations (see Figure 5.6). To deal with this problem additional hardware unit or software fix is required. In the Fischer proposal the output stage can possible be modified for effective addition/subtraction, however, the custom made “editing logic”, is expected to become very complex. In addition, none of the above proposals presented how they will detect the $N2 > N1$ (BCD) situation. Our design does not require such additional effort or resources with some increase in latency and area.

In terms of latency the “best” design is the one proposed by Haller. Compared to it, our proposal is 21% slower, but uses 15% to 16% less hardware resources (LUTs and Slices). When considering area, Hwang’s proposal ranks best, but have the deficiency as described above. In general, our design is between 11% and 21% slower than the others. Assuming a design with a latency of 10ns, e.g. Haller, (worse case for our design) and a detection of $N2 > N1$ (BCD) “for free”, the average execution time is going to be similar to ours (12ns) when 20% of the operations require the compensation cycle as introduced earlier.

When absolute performance is considered, our design achieves 82.6 MOPS (1/12ns). Please note that this will be significantly improved when a pipelined version of the design is considered. The latency of the carry-propagate-adder is expected to determine the operating frequency in such a case.

5.4 Conclusions

This chapter provides the details for a novel adder/subtractor arithmetic unit that combines Binary and Binary Code Decimal (BCD) operations in a single structure. The unit is able to perform effective addition-subtraction operations on unsigned, sign-magnitude, and various complement representations. Our

proposal can be implemented in ASIC as a run time configurable unit as well as in reconfigurable technology as a run-time reconfigurable engine. Under the assumption that significant part of the hardware in the proposed structure is shared by the different operations reconfigurable platforms with partial reconfiguration become an interesting target. The proposed unit and several widely used adder organizations were synthesized for 4vfx60ff672-12 Xilinx Virtex 4 FPGA for comparison. Our design achieves a throughput of 82.6 MOPS with similar area x time (AxT) product when compared to the other proposals, using only 2% of the available resources of the targeted FPGA.

Chapter 6

Address Generator for complex operation Arithmetic Units

Nowadays, performance gains in computing systems are achieved by using techniques such as pipelining, optimized memory hierarchies [98], customized functional units [99], instruction level parallelism support (e.g. VLIW, Superscalar) and thread level parallelism [100] to name a few. These time and space parallel techniques require the design of optimized address generation units [101–104] capable to deal with higher issue and execution rates, larger number of memory references, and demanding memory-bandwidth requirements [105]. Traditionally, high-bandwidth main memory hierarchies are based on parallel or interleaved access of parallel memories. Interleaved memories are constructed using several modules or banks. Such structures allow distinct banks access in a pipelined manner [106]. In this chapter we propose an address generation unit (AGEN) for efficient utilization of n -way-interleaved access to a parallel main memory containing vector data, e.g. for operate with arithmetic units that supports complex operations and kernels like SAD (sum of absolute differences) and MVM (matrix-vector multiply). More specifically, the main contributions of this chapter are:

- An AGEN design capable of generating 8 x 32-bit address in a single cycle. In addition, arbitrary memory sequences are supported using only one instruction.
- An organization that uses optimized Boolean equations to generate the 8 offsets instead of an additional adder's stage.
- An FPGA implementation of the proposed design able to fetch 1.33 Giga

operands per second from an 8-way-interleaved memory system using only 3% of the targeted device.

We note, that in this chapter we use number representation notations different from Chapters 2, 3 and 4. The reasons are the same as explained in Chapter 5, section 5.1.1.

The remainder of this chapter is organized as follows. Section 6.1 outlines the necessary background on parallel and interleaved-memory systems. Section 6.2 we considered a vector architecture, the memory interleaving mechanism and the design of the AGEN Unit. In Section 6.3, we discuss the experimental results in terms of used area and latency. Finally, in Section 6.4 conclusions and future work are presented.

6.1 Background

The use of multiple memory banks to provide sufficient memory bandwidth is a common design approach when high memory performance is required [107]. The fixed addressing distance between two consecutive elements to be accessed is called a stride. The stride describes the relationship between the operands and their addressing structure. A parallel memory, organized in several banks, which store elements in a strode manner and which employs a multiplexed access is called interleaved memory [108, 109].

Given that an n -bit address memory field can be divided into 1) memory-unit-number and 2) address in memory unit (memory-address), two main addressing techniques arise from this basic address division as depicted on Figure 6.1.

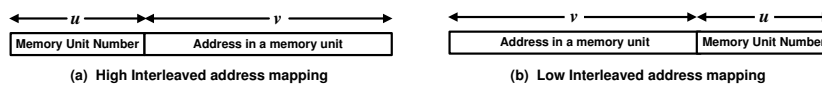


Figure 6.1: Interleaved memory formats.

- (a) *High interleaved* addressing mapping utilizes the low address bits v as memory-address in the unit, while the higher bits u represent the memory-unit-number. This technique is used by the traditional scalar processors (GPP) with multiple memory pages.
- (b) *Low interleaved* memory mapping use the low address bits u to point out the memory-unit-number, while the higher memory bits v are the memory-address.

Figure 6.2 (a) presents a time multiplexed memory (Interleaved Memory) and Figure 6.2 (b) a space-multiplexed memory (Parallel Memory). In a interleaved memory, (consider a_1, a_2, a_3 in Figure 6.2(a)), each memory module returns one word per cycle. In parallel memories, it is possible to present different address to different memory modules, as suggested in Figure 6.2(b), so that a parallel access of multiple words can be done simultaneously. Both the parallel access and the pipelined access are forms of parallelism employed in a parallel memory organizations [108]. In this thesis we consider a parallel memory organization similar to the one presented in Figure 6.2 (b), with elements distributed with odd stride.

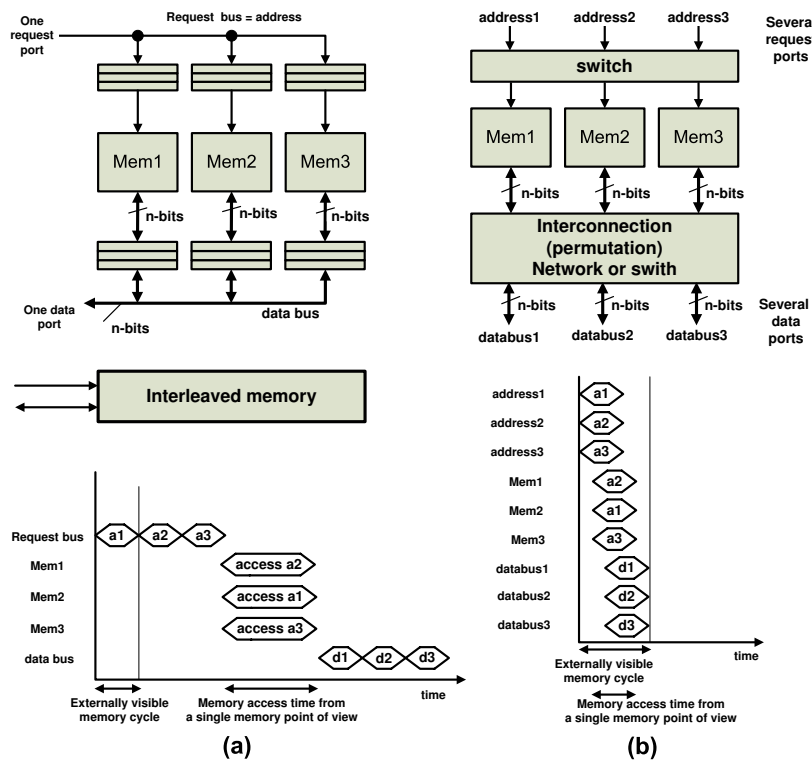


Figure 6.2: Main accumulator circuitry

Low interleaved-address mapping, such as the memory system with 8 banks and data structure with $stride = 1$ presented in Figure 6.3, has their data distributed in a round-robin like fashion among the memory banks. For example, word 0 is stored in bank 0, word 1 is stored in bank 1. In general, word x is located in bank $x \text{ MOD } 8$. In this figure, one *Major Cycle* (memory latency) is

subdivided in 8 *Minor Cycles*. The hybrid accessing presented in this figure, allows the retrieving of 8 consecutive elements in one *Major Cycle* and 7 additional *Minor Cycles*. This is due to the fact that the eight consecutive elements from the memory banks are retrieved in parallel. Those read values are stored in intermediate data registers from which they are issued to the functional units in a pipelined manner in this model (using 7 additional *Minor Cycles*). With this memory architecture the retrieving of x single-word elements will take $Major\ Cycle + (x - 1) Minor\ Cycles$ in the best case scenarioⁱ.

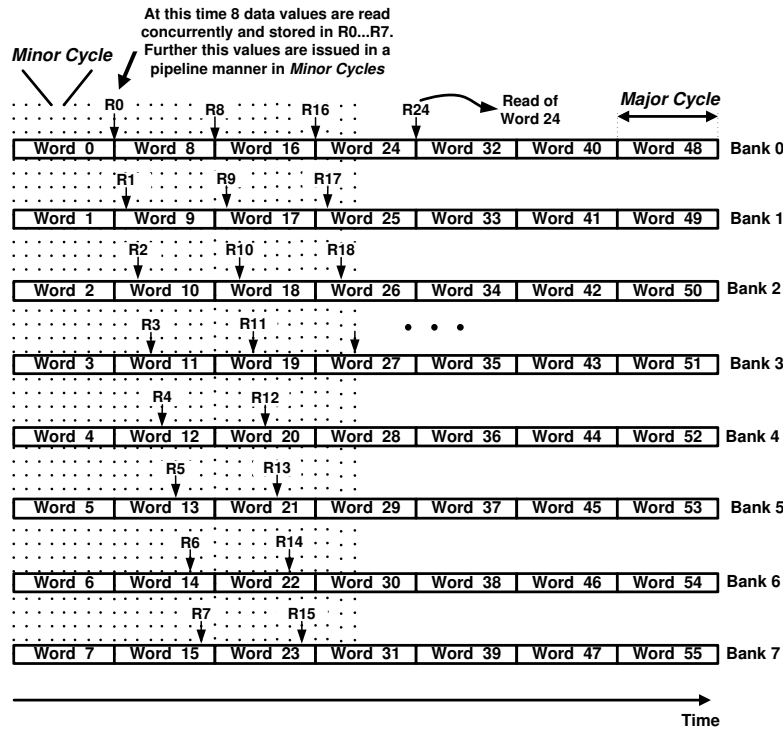


Figure 6.3: Interleaved memory pipelined access to memory

6.2 AGEN Unit Design

We consider a vector co-processor consisting of a group of reconfigurable functional units [6, 110] coupled to a core processor. Figure 6.4 presents

ⁱDepending in the latency of the organization e.g. switch, the *Major Cycle* can be divided into a limited number of $(x - 1) Minor\ Cycles$

this organization. An arbiter is used to distribute the instructions between the vector-coprocessor unit and the GPP following the paradigm proposed in [16]. Please note that many current platforms implement similar approaches, e.g. the Fast Simplex Link interface and the Auxiliary Processor Unit (APU) controller for MicroBlaze and PowerPC IP cores [111]. For the parallel memory block of Figure 6.4, we consider a similar design approach presented in Figure 6.2.

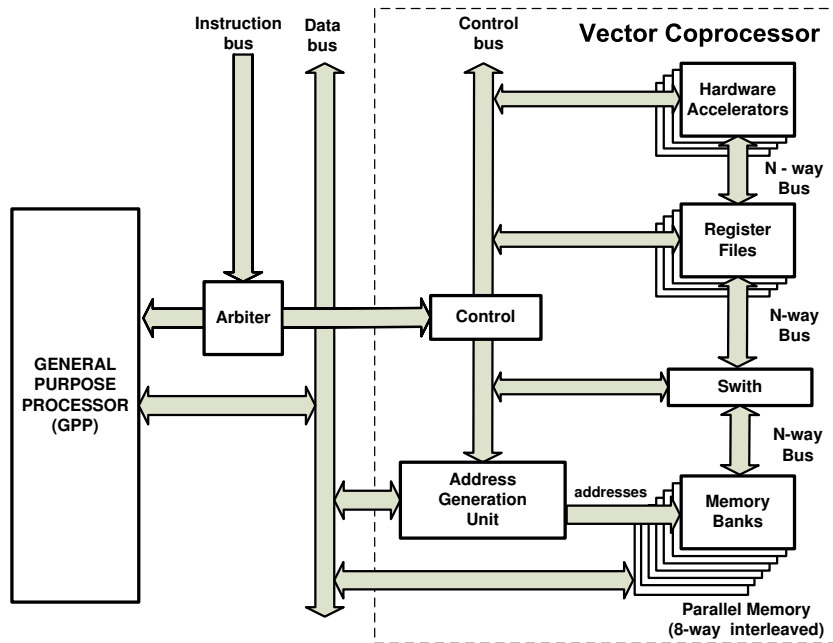


Figure 6.4: Block diagram of the reconfigurable custom computing unit [4]

The memory banks presented in Figure 6.4 are built using dual ported memories, e.g. BRAMs [112] in case of FPGA implementation, shared by both processors, the GPP and the vector. One port of the BRAM is used by the GPP processor as a linear array memory organization with *high interleaved* address mapping. The second port is used by the vector-coprocessor unit. The memory access from the vector-coprocessor side requires dedicated AGEN unit (different from the one embedded into the core processor) that generates the addresses for the 8-way low interleaved memory organization in the correct sequence order. The vector data is distributed in an interleaved-way, scattered by the stride values, that requires 8 different addresses for each memory ac-

cess. The AGEN unit is configured to work with single or multiple groups (with the same stride) of streamed data using a single instruction. The AGEN special instruction configures the base addresses, the stride and the length of the particular streaming data format. The memory accesses can be performed in parallel with the execution phase of a previous iteration using the decoupled approach as presented in [99].

6.2.1 Memory-Interleaving Mechanism

In this section the mechanism to retrieve n data elements in parallel is presented. Figure 6.5, shows eight different stride cases, with odd strides ≤ 15 for eight memory banks. For example, the stride shown in case (b), is three. One can see, that in all of the cases the data is uniformly distributed in the memory banks. This fact suggests the feasibility of an AGEN capable to generate the effective addresses of n data elements every major cycle. This can be formally stated as follows:

n data elements stored in n memory banks can be retrieved in a single major cycle if the stride is an odd integer and n is a power of two.

Otherwise stated this can be extended as follows:

n data elements stored in n memory banks can be retrieved in a single major cycle if $\gcd(n, \text{Stride})=1$.

The notation $\gcd(a, b)$ is used for the greatest common divisor. Two integers a, b are relatively prime if they share no common positive factors (divisors) except of 1, e.g $\gcd(a, b) = 1$.

Extension to the general case: Let's consider n banks of memory each holding m memory cells. The $m \times n$ memory array can be represented as a matrix $[m \times n]$ where each column corresponds to a memory bank. In addition, the cell i of the memory bank j corresponds to the matrix element with indexes (i, j) . We denote this matrix as A and consider $n = 2^h$ and m for its dimensions, with $h, m \in \mathbb{N}$. In addition, the stride of the data structures stored on the memory is an integer $\text{Str} = 2q + 1, q \in \mathbb{N}$.

From now on, the data stored in the memory banks will be considered as matrix A elements. Let the n consecutive data elements placed in different memory

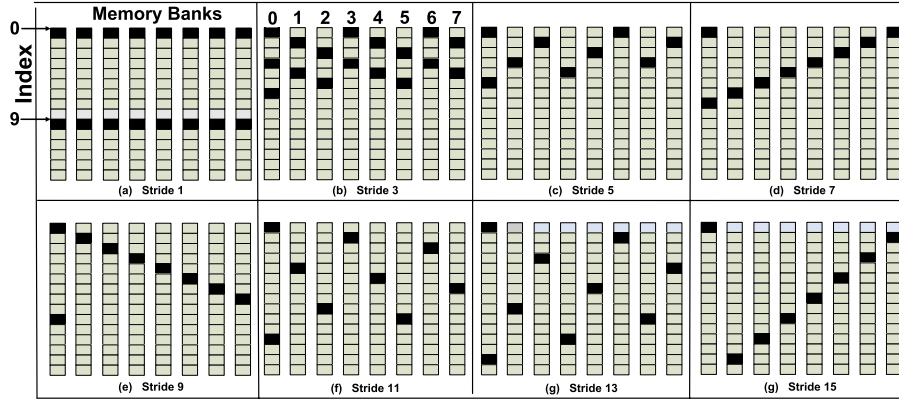


Figure 6.5: An 8-way interleaved memory banks with odd strides ≤ 15 .

banks be denoted by:

$$a_0, \dots, a_{n-1}. \quad (6.1)$$

Remark 1. Every element a_α , with $\alpha = 0, \dots, n-1$, is identified in the matrix by its row-index i , with $i = 0, 1, \dots, m-1$, and its column-index j , with $j = 0, 1, \dots, n-1$. This means that there exists a one-to-one relation among a_α and the indexes pair (i_α, j_α) . Additionally, the pair (i_α, j_α) can be used to represent a_α as a number in base n , obtainable as juxtaposition of i_α as most significant digit and j_α as least significant digit. The two indexes can also be used in a base 10 representation. Therefore, we have the following chain of equivalent representations for a_α :

$$a_\alpha \leftrightarrow (i_\alpha, j_\alpha) \leftrightarrow (i_\alpha j_\alpha)_n \leftrightarrow (ni_\alpha + j_\alpha)_{10}. \quad (6.2)$$

As an example, Table 6.1 shows the chain of representations as defined in equation (6.2) for a case where $n = 8$ and $\text{Str} = 3$.

Remark 2. Without loss of generality, we can assume that the first element a_0 stored in the matrix remains at position $(i_0, j_0) = (0, 0)$.

Lemma 1. The number of rows necessary to hold n elements with stride $\text{Str} = 2q + 1$, $q \in \mathbb{N}$ is Str .

Table 6.1: Correspondence $a_\alpha \leftrightarrow (i_\alpha, j_\alpha) \leftrightarrow a_{\alpha|n} \leftrightarrow a_{\alpha|10}$ for $n = 8$ and $\text{Str} = 3$.

Element a_α	Row-Index i_α	Column-Index j_α	$a_{\alpha 8}$	$a_{\alpha 10}$
a_0	0	0	00	0
a_1	0	3	03	3
a_2	0	6	06	6
a_3	1	1	11	9
a_4	1	4	14	12
a_5	1	7	17	15
a_6	2	2	22	18
a_7	2	5	25	21

Proof. The number of cells ($\#_{cell}$) necessary to store n elements with stride Str is $\#_{cell} = n + (\text{Str} - 1)n = n(2q + 1)$. Therefore, the number of rows is

$$\#_{cell} \bmod n = n(\text{Str}) \bmod n = \text{Str}. \quad (6.3)$$

□

Remark 2 and Lemma 1 imply that the necessary rows to store the n elements with stride Str are:

$$\{0, 1, \dots, \text{Str} - 1\} \quad (6.4)$$

The n data a_α can be defined recursively. If $a_0 = (i_0, j_0)$ the elements a_2, \dots, a_{n-1} can be recursively defined as follows:

$$a_\alpha = a_{\alpha-1} + \text{Str}. \quad (6.5)$$

Theorem 1. Let n be the number of elements a_α , with $\alpha = 0..n - 1$, stored in a matrix A , $m \times n$, with $n = 2^h$. Let the stride be the integer $\text{Str} \in \mathbb{N}$. If (i_α, j_α) and (i_β, j_β) are the couples of indexes identifying a_α and a_β in the matrix and $\text{gcd}(n, \text{Str}) = 1$, we have:

$$j_\alpha \neq j_\beta \quad \forall \alpha, \beta \in [0, \dots, n - 1], \alpha \neq \beta. \quad (6.6)$$

Proof. Without loss of generality, by Remark 2, we can assume $(i_0, j_0) = (0, 0)$. By contradiction let $j_\alpha = j_\beta$. We have two possible cases: (1) $i_\alpha = i_\beta$ and (2) $i_\alpha \neq i_\beta$.

The first case is not possible: more precisely, if $i_\alpha = i_\beta$ will lead to $a_\alpha = a_\beta$ since $j_\alpha = j_\beta$ (see Remark 1).

In the second case: $i_\alpha \neq i_\beta$. Firstly, by equation (6.4), it follows:

$$i_\beta - i_\alpha \in [0, \text{Str} - 1]. \quad (6.7)$$

Without loss of generality we can assume $\beta > \alpha$. By (6.5) we have:

$$a_\beta = a_{\beta-1} + \text{Str} = a_{\beta-2} + 2\text{Str} = \dots = a_\alpha + x\text{Str}, \quad (6.8)$$

with $x \in \mathbb{N}$ and $x < n$; it is straightforward to show that $x = \beta - \alpha$. By using the representations in base 10 of a_α and a_β (see (6.2)), the equation (6.8) becomes:

$$ni_\beta + j_\beta = ni_\alpha + j_\alpha + x\text{Str}, \quad (6.9)$$

taking into account the assumption $j_\alpha = j_\beta$ we can rewrite (6.9) as

$$n(i_\beta - i_\alpha) = x \text{Str}. \quad (6.10)$$

Since $\text{gcd}(n, \text{Str}) = 1$ and n divides the product $x \text{Str}$, it follows that n is a divisor of x . This implies that: $x = r n$, with $r \in \mathbb{N}$. Therefore $x > n$ which contradicts the original hypothesis. As a consequence, it must be that $j_\alpha \neq j_\beta$, for all $\alpha, \beta \in [0, \dots, n - 1]$.

□

Remark 3. The previous theorem can be reformulated saying that *if n data elements are stored in n memory banks with a fixed stride Str and the $\text{gcd}(n, \text{Str}) = 1$, each data element is stored in a different memory bank.*

Corollary 1. By Theorem 1 it follows that the data are stored in different memory banks if $n = 2^h$ and Str is an odd integer and viceversa if n is an odd integer and $\text{Str} = 2^h$.

Example: Let's consider the case (b) presented in Figure 6.5. In this example, $n = 8$, the $\text{Str} = 3$. This is also the case considered in Table 6.1. Column 3 of Table 6.1, shows that each element of this data structure belongs to a different column and therefore to a different memory bank. This follows by Theorem 1. If there exist two elements a_α, a_β with the same column index then there exists $x < 8$ such that: $n(i_\beta - i_\alpha) = x(2q + 1)$ ($q = 1$ in this case). Considering that $n = 8$ in our example, $n(i_\beta - i_\alpha)$ can be either 8 or 16. The difference cannot be 0 since in that case $i_\alpha = i_\beta$ and therefore $a_\alpha = a_\beta$. As a consequence, we have two cases $8 = 3x$ or $16 = 3x$ and both equations don't have an integer solution for x .

6.2.2 The AGEN Design

The effective address computation is performance-critical (see e.g. [113]). The AGEN unit described in this section generates **eight** addresses for fetching data elements simultaneously from an 8-way interleaved memory system at high speed. The AGEN is designed to work with multi-operand units [6, 110] and uses a special-purpose-instruction such as the ones presented in [114]. In Figure 6.6 an example of such instruction is presented. The multiple base addresses in this instruction are necessary for cases with multiple indices such as SAD and MVM operations.

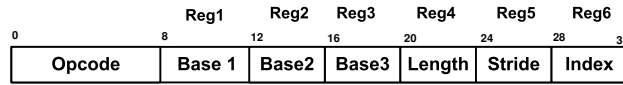


Figure 6.6: Compound instruction

The 4-bit instruction fields depicted in Figure 6.6, define the registers containing the addresses and/or the length and the stride parameters of the data structure to be accessed. More precisely they are:

- $Base_i (i = 1, 2, 3)$. These registers contain the physical memory address that point to the first elements of an data arrays to read or write in the interleaved memory organization. For example, the minuend and subtrahend in the sum of absolute differences (SAD) instruction or multiplicand, multiplier and addendum in multiply-accumulate (MAC) operations.
- *Length*. This register holds the number of n -tuples (cycles) needed to gather y -elements from the memory. For example, when length value is 10 and $n = 8$, 80 elements will be retrieved in 10 memory accesses.
- *Stride*. This register holds the distance between two consecutive data elements in an n -way interleaved memory. In our case the possible strides are odd numbers in the range between 1 and 15. Thus, strides are expressed as $2q + 1$, with $0 \leq q \leq 7$. In our design, the eight possible stride values are encoded using three bits of the available four in the compound instruction.

- *Index*. The address stored in this register has two uses:
 - The register contains the vertical distance between two consecutive groups of n elements. For example, Figure 6.5 (a) presents the index (also referred as vertical stride) that is equal to 9.
 - Sometimes the AGEN can be used to retrieve a single data word. In this case, the register value is used as an unsigned offset address in a Base + offset scheme of addressing.

Equation (6.11) describes the effective address (EA) computation. EA is obtained by the addition of the base-stride (BS) value, the index (IX) value, and the memory-bank offsets. These offsets are represented by $A_i(0..3)$ addresses, necessary offset for generating the 8 addresses in parallel (see Figure 6.7(c) and Table 6.2). Figure 6.7(e) depicts the $8 \times$ EA generators for the targeted 8-way interleaved memory system.

$$EA_i = BS + A_i(0..3) + IX \quad \forall 0 \leq i \leq 7 \wedge RES \geq 0 \quad (6.11)$$

where $RES \geq 0$ suggest that there are still elements to be processed and the counter of elements depicted on Figure 6.7 (c) does not reach the zero value.

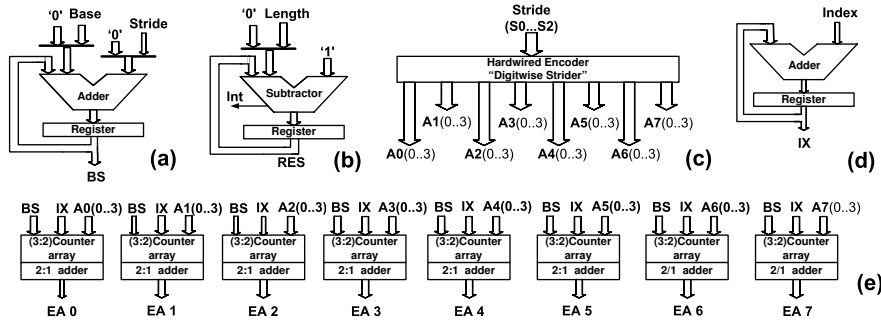


Figure 6.7: Address generation unit: (a) Accumulator for BS computing, (b) Accumulator for loop control, (c) Hardwired encoder, (d) Index accumulator, (e) Final addition effective address computing

The first addendum term (BS) of (6.11) is computed using the following relation: $BS = Base + k \text{ Stride}$. During the first cycle, BS is equal to the base address, therefore a 0 value is used for the second term. Thereafter, the stride offset is added for each k iteration. Note that the stride value is equal to the offset between two consecutive data elements in the same column as was presented in Section 6.2.1 (see also Figure 6.5). As mentioned earlier,

Figure 6.7(b) depicts the subtractor used for counting the number of memory accesses. In each clock cycle, e.g., equivalent to 8 iterations of an unrolled loop, the subtractor value is decremented by one until it reaches zero. A negative value of the subtractor result (underflow) asserts the “Int” flag, indicating the end of address generation process. Figure 6.7(c) represents the logic block for computing the offset-value $A_i(0..3)$, which will be discussed in the address transformation subsection in more details. Finally, Figure 6.7(d) shows the IX computation.

The accumulator structure presented in Figure 6.7 (a) is composed by two stages partially (4-bits only) shown in Figure 6.8. The first stage consists of an (4:2)counter which receives the SUM and the CARRY signals of the previously computed value. The other two inputs (shown left on the figure) receive the mux-es outcomes used to select the appropriate operands (base and stride values) as explained above. The second stage consist of a (2:1) adder that produces the BS values. This pipelining allows a low latency in the generation of the addresses.

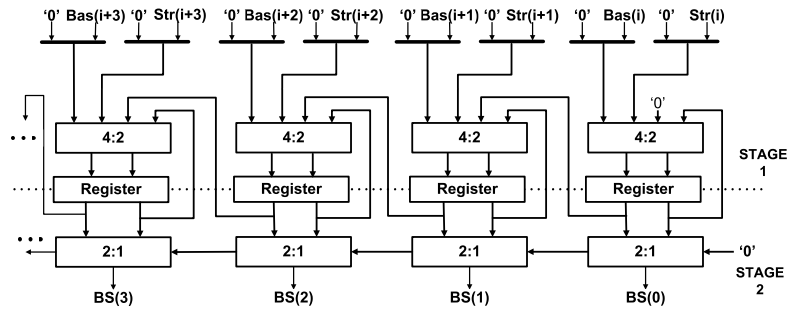


Figure 6.8: Main accumulator circuitry

Address transformation: The stride values supported by our implementation are encoded using 3 bits represented by $S_2S_1S_0$. The pattern range $000_2..111_2$ encodes the $2q + 1$ stride values with $0 \leq q \leq 7$, encoding 8 different odd strides. A hardwired logic is used to transform the encoded stride values into the corresponding $A0_{(0..3)}, \dots, A7_{(0..3)}$ address offsets using a memory-bank-wise operation. A “memory-bank-wise” address is created based on the stride value. For example, consider Figure 6.5 (c) that presents

the case for stride = 5. In this case, concerning banks 1 and 4 offset values of 3 and 2 are required. These correct memory-bank-wise values are generated by our hardwired logic. Please note that our approach supports all possible odd stride values in the range between 1 and 15. The exact transformations are presented as a set of equations in Table 6.2 for the 8 necessary offsets for a parallel access.

Table 6.2: Hardwired encoder - Set up table of equations

Bank	A_0	A_1	A_2	A_3
0	0	0	0	0
1	$S_2 \cdot \overline{S_1} \cdot \overline{S_0} + \overline{S_2} \cdot \overline{S_1} \cdot S_0 + S_2 \cdot S_1 \cdot S_0 + \overline{S_2} \cdot S_1 \cdot \overline{S_0}$	$S_2 \cdot \overline{S_1}$	$S_2 \cdot S_0 + S_1 \cdot S_2$	$S_2 \cdot S_1$
2	S_1	$\overline{S_2} \cdot \overline{S_1} \cdot S_0 + S_2 \cdot \overline{S_0} + S_2 \cdot S_1$	$\overline{S_2} \cdot S_1 \cdot S_0$	$S_2 \cdot S_0$
3	S_2	$S_2 \cdot \overline{S_0}$	$\overline{S_2} \cdot S_1$	$S_2 \cdot S_1$
4	S_0	S_1	S_2	0
5	S_2	$\overline{S_2} \cdot S_0$ $\overline{S_2} \cdot S_0$	$S_2 \cdot \overline{S_1} \cdot \overline{S_0} + S_2 \cdot S_1 \cdot S_0$	$S_2 \cdot \overline{S_1} \cdot S_0$
6	S_1	$S_2 \cdot S_1 + S_2 \cdot S_0 + \overline{S_2} \cdot S_1 \cdot \overline{S_0}$	$S_2 \cdot \overline{S_1} \cdot \overline{S_0}$	$S_2 S_1 \overline{S_0}$
7	$S_2 \cdot \overline{S_1} \cdot \overline{S_0} + S_2 \cdot S_1 \cdot S_0 + \overline{S_2} \cdot S_1 \cdot \overline{S_0}$	$S_2 \cdot \overline{S_1}$	$S_2 \cdot \overline{S_1} + S_2 \cdot S_0$	0

e.g. the address bit A_2 for bank 1 will be: $A_2 = S_2 \cdot S_0 + S_1 \cdot S_2$.
This value (offset) is added to the current Base address value for obtain EA

6.3 Experimental Results Analysis

The proposed address generation unit was described using VHDL, synthesized and functionally validated using ISE 7.1i Xilinx environment [115]. The target device used was VIRTEX-II PRO xc2vp30-7ff1696 FPGA. Table 6.3 summarizes the performance results in terms of delay time and hardware utilization. The proposed pipelined organization has a bottleneck stage that requires 6 ns (166.6 MHz) for processing the addresses. We also present the latencies of the other embedded major sub-units used in our proposal.

Table 6.3: The address generation unit and embedded arithmetic units

Unit	Time delay (ns)			Hardware used	
	Logic Delay	Wire Delay	Total Delay	Slices	LUTs
Address Generation Unit	4.5	1.4	6.0	673	1072
Hardwired encoder (Digitwise) ‡	0.3	-	0.3	9	16
(4:2)Counter ‡	0.5	0.5	1.0	72	126
(3:2)Counter ‡	0.3	-	0.3	37	64
32-bit CPA (2:1) adder ‡	2.2	0.7	2.9	54	99

‡: Embedded circuitry into AGEN unit. Those are presented without I/O buffers delays.

The 6 ns of latency presented in Table 6.3 for the Address Generation Unit, corresponds to the latency of the 32-bit CPA adder employed to build the second stage of the main accumulator circuit. This bottleneck can be improved using a deeper pipeline unit when it is considered the construction of the CPA (see [116] for details). A pipelined CPA will improve the overall performance of the proposed unit. The last is important for technologies with lower memory latency like the Virtex 4 and Virtex 5 devices [117] that can require fastest AGENs that operate on higher frequencies than the 166 MHz provide by the current solution. The AGEN unit proposed here uses 3 stage pipeline. The first two pipeline stages correspond to the accumulator for the BS computation (Figure 6.7(a)) and the third one to the (3:2)counter array and the final stage (2:1) adder. The latter constitutes the critical path for our implementation.

In summary, the proposed AGEN unit reaches an operation frequency of 166 MHz. Otherwise stated, our proposal is capable to generate 1.33 Giga addresses of 32-bits (totaling 43.5 Gbps) from an 8-way interleaved memory. Concerning the silicon area used by the proposed AGEN, the total unit uses only 3 % and 4 % of the targeted device in terms of slices and LUTs respectively.

6.4 Conclusions

A detailed description of an efficient vector address generation circuitry for retrieving several operands from an n -bank interleaved memory system in a single machine cycle was presented. The proposal is based on a modified version of the low-order-interleaved memory approach. The theoretical

foundation of the proposed approach that guarantees the trivial indexing structure was also presented. Moreover, a new AGEN unit capable to work with dedicated multi-operand instruction that describes inner loops was introduced. An analysis of the latency of the proposed unit indicates that it is capable to generate 8×32 bit addresses every 6 ns. In addition, our design uses only 3 % of the hardware resources of the targeted FPGA device.

Chapter 7

Comparative Evaluations

In this chapter, we evaluate the four designs of arithmetic accelerators presented in the previous chapters, namely “Arithmetic Unit for collapsed SAD and Multiplication operations (AUSM)”, “AUSM Extended”, “Fixed Point Dense and Sparse Matrix-Vector Multiply Arithmetic Unit” and “Arithmetic unit for Universal Addition”. We consider FPGA technology for our evaluations. This chapter is organized as follows: in Section 7.1, the silicon area utilized and the critical paths (latency) of the proposed arithmetic accelerators are compared to the most relevant related works. In Section 7.2, we discuss the results of integrating a complete system consisting of the SAD accelerator as a coprocessor and a Microblaze processor as a GPP. Finally, the chapter is concluded in Section 7.4.

7.1 Arithmetic Accelerators

The results obtained from the synthesis of our arithmetic accelerators on Xilinx Virtex II PRO technology as well as the obtained simulation data are compared with related works. More specifically, we have considered area utilization and latency.

7.1.1 The SAD Accelerator

Assuming full search scheme in video encoding applications, the number of all candidate blocks are expressed by the following equation:

$$q = (2p + 1)^2 \quad (7.1)$$

where the search is limited to a maximum displacement p , and q represents the number of candidate blocks [43]. A motion estimation technique like the sum of absolute differences that operates on frames of $N \times N$ pels will require N^2 subtractions, magnitude operations and additions of absolute values of each candidate block. The area of search is $(2p + N)^2$, thus, for a block of N^2 pels, a block search will require retrieving a number of words denoted by the following equation:

$$(2p + N)^2 + N^2 \quad (7.2)$$

The Common Intermediate Format (CIF) defines the resolution of a frame for video-conferences which is 352x288 in PAL or 352x240 in NTSC. This format is considered in our work as a base parameter to measure the throughput of an accelerator. The processing of 30 frames per second (fps) for a CIF PAL frame with $p = 8$ using full search and block matching algorithm (BMA) will require 2.6 Giga operations per second (GOPS). In addition, a bandwidth of 243 Mbps between the memory and the accelerator will be required for an 8-bit pel representation. Those large numbers of computational operations were fulfilled with different related acceleration works that principally include systolic arrays structures, contrary to our approach with multi-operational arithmetic units. In the following subsection, a group of important related works are examined.

Comparison to Related Works

An accelerator constructed with an array of 16 PEs that computes the difference and absolute value, augmented with a tree of adders for reducing the partial results were presented in [118]. This organization implemented over an APEX EP20K200EQC240-1 FPGA running at 120 MHz, computes 29296 (16×16) blocks per second equivalent to processing a CIF in 13.5 ms.

An improvement to this performance is achieved in the proposal presented in [119], designed for working over portable devices. This solution employs 65 % of the VIRTEX II XC2V3000-4 device for setting up 10 Motion Estimation Units (MEUs) and is capable to process one CIF video frame in 8.712 ms using a clock frequency of 93 MHz.

A systolic array architecture that uses a group of PEs to compute the absolute value, while their summation is computed using the *online arithmetic adder* is proposed in [56]. This proposal requires 489 slices for set up a customized solution for a 4×4 macro-block over a Virtex II PRO FPGA, achieving a throughput of 26.56 millions of SAD computations per second and up to 249 CIF frames per second.

Another solution that uses reconfigurable technologies was presented in [55]. This organization contains 16 PEs; each PE is composed by one subtracter, one adder/subtracter and two registers R1 and R2. Register R1 is used to store the partial computed value (into an accumulator structure), and register R2 is used to store the final computing value every fourth cycle. This sequential computing requires 16 cycles to calculate a 4×4 macro-block and is able to process up to 1479 CIF video frames per second using 14500 slices and 28.500 LUTs when operating at 150 MHz. The processing of the data over the subtracter unit and a regular accumulator (with local feedback) in a sequential way has been proposed in several works; those proposals achieve a similar performances than the aforementioned ones, see e.g. [120, 121].

On the other hand, our proposal has an organization that calculates several SAD values concurrently. The arithmetic accelerator we have proposed in Chapter 2 collapses an equivalent of 8 SAD units, computing concurrently the absolute values of the 8 differences (half 4×4 macroblock) and at the same time the 7 concurrent operands additions in 25 ns using a parallel memory of 256 bits. For our experiments, we consider a Virtex II PRO-6 device, allowing a total throughput of 320 mega SAD operations per second. Table 7.1 presents the latency for processing a CIF in ms, the # CIFs per second that can be processed and the hardware cost in slices used by the examined related work and in our proposal (see Chapter 2 and 3). We use the data reported in [55, 56, 118, 119], focusing on the SAD units embedded in those proposals. We have considered in this estimation that those units have the sufficient bandwidth to receive the data.

Table 7.1: Area and latency used to process one CIF

Proposal	Units	Frame processing time (ms)	Cost (Slices)	CIF (PAL) per second	Frequency MHz
Sayed [119] §	10 Motion estimation PE units	8.7	14336	114	93
Amer [55] §	16 parallel PE serial organization	0.676	14500	1479	150
Olivares [56]	Online arithmetic systolic array	3.81	480	263	197
Wei [118] ξ	One dimensional Systolic array	13.5	1640	74	120
Ours	Collapses of multiple additions	0.317	1861	$394 \times 8 = 3156$	40

In Table 7.1 “ ξ ” indicates that an Altera APEX EP20K200EQC240-1 translation [122] was made for a comparison with other proposal implemented over Xilinx FPGAs; and “ ζ ” denotes that the number of the silicon area used includes: the accelerator, the address generation unit, the memory elements and multiplexers used to route the data.

The cost in slices of our proposal presented in the above table considers the total cost of the vector coprocessor for the SAD unit in order to make a fair comparison (see section 7.2 for details). The data in Table 7.1 suggest that our design is 2.13 faster when compared with [55], a proposal that uses 16 PEs. On the other hand, our proposal consumes more hardware, 3.78 times compared with [56] and 1.14 times compared with [118]; nevertheless, our proposal achieves considerable speed-ups of 12 and 42 respectively. Our design outperforms the considered works at the cost of relatively small hardware penalty, using 8x32 bits wide parallel memory.

7.1.2 The Dense and Sparse Matrix-Vector Multiply Unit

In this section, we evaluate our dense and sparse matrix-vector multiplication unit presented in Chapter 4. We compare our design to some related solutions which use massive parallel SIMD organizations.

Dense Matrix Multiplication

A comparison of some important implementations over reconfigurable technologies is presented in [123]. We use this analysis to differentiate our proposals from the related work. Table 7.2 presents the hardware used by different organizations, expressed in CLB, and the latency required for processing a 4×4 matrix multiplication. Contrary to other approaches that employ systolic structures, our proposal is based on the collapsing [39, 40] of several multiplication operations and a group of multiple-addition hardware. Those arithmetic resources were arranged in an array of counters, maintaining the CARRY-SUM representation until the final addition. Thus, the total latency of the whole arithmetic unit is reduced (see Chapter 4). Note, that other proposals require the carry propagation at least two times, because they are based on the sequential concatenation of arithmetic units. Furthermore, some of those proposals are based on the use of a sequential accumulator organization. In our case, we collapse the operations (multiplication and multiple-additions) requiring only one carry propagation in the final (2:1) adder.

Table 7.2: Dense matrix multiplication comparison - Related work

Proposal	Units	Latency (ns)	Cost (CLBs)
Mencer [123, 124]	Bit serial Multipliers (Booth encoding)	570	477
Amira [123, 125]	Pipelined MAC unit. The last stage has a feedback path with a (4:2)counter for speed up the multiple accumulation operation.	336	296
Jang [123]	Linear Systolic Array, each PE has a multiplier and adder to process with a feedback the MAC operation.	150	390
Prasana [123, 126]	Linear systolic array with n processors with internal storage, ALUs and multiplexers to route the data	150	420
Ours	Collapsed multiply and multiple addition unit	149	1140

All the works are compared under the same technology, we approximate as stated in [123]. Our proposal was re-targeted to a Virtex XCV1000 device for cost (CLBs) estimation purposes.

From Table 7.2, it is evident that our proposal is as fast as the best performing related designs. Indeed, our unit requires more hardware resources, but this is an extra cost we are paying for the added functionality for sparse matrices.

Sparse Matrix Multiplication

Linear arrays, rectangular arrays, hexagonal arrays and other structures with different grade of connectivity between the PEs operating in a synchronous way [127, 128] were proposed as a solution to speed up the sparse matrix-vector multiplication. For example, Ogielski et al. [129] presents a study of the matrix-vector computation on a rectangular processor. The idea to “Load balancing” (efficient assignment) of non-zero-elements (NZE) were proposed and tested over MasPar MP-1216 built of 16,348 RISC processors (128x128 array) which achieved a performance of 48 Mflops (Similar massive approaches are analyzed in [130]). The referred above micro-organizations carry out the following actions for each row:

1. Fetch the required vector elements $A_{(i,j)}$;
2. Perform the local multiply operation $P_{i,j} = A_{i,j}x b_j$
3. Add the partial products sequentially $C_i = \sum C_{i,j}$

4. Store the result.

The distribution of the NZE were studied in [76, 77, 131, 132] and several algorithms were proposed. Zhuo and Prasanna's FPGA solution [77] propose also the sequential iteration over dot products, parallelizing each product and sequentializing the addition. This approach follows a previous proposal of the same author for dense matrix processing [123, 126].

In contrast to the approaches presented, our solution uses a simple hardware unit for distributing dynamically the NZEs into the multiplication and the multiple addition units (see Chapter 4). Our approach does not incur any overhead due to splitting the NZE of a particular row. We make that distribution dynamically. Furthermore, we do not have any overhead due to the communication between the PEs. The overhead, introduced by managing the irregular distribution of the NZE is low, e.g., about 1.8 ns (see Chapter 4). The latency values presented on Table 7.2 are also hold for the sparse matrix multiplication.

7.1.3 The Binary and Decimal Adder

Decimal arithmetic applications [84] require the speed up of the BCD addition/subtraction operations. Early solutions proposed by the academia and industry include binary arithmetic units that support BCD manipulation with slight modifications of the basic ALUs. The creation of "dedicated units" [86, 87] was a step further into the research of units with better performance when decimal arithmetic is considered. Units that collapse binary and BCD functionalities were proposed in [33, 84, 133, 134] introducing important hardware reutilization for the support of several functionalities. Nevertheless, the main drawback of those proposals is their inability for efficient processing with effective addition/subtraction characteristics, the way our proposal does. Previous proposals require the use of an additional processing cycle to achieve correct results when operating with signed-magnitude representations. Figure 7.1 presents the percentage of operations with subtrahend greater than the minuend required to make our arithmetic accelerator better than the fastest related work [96] (see Chapter 5). It has been observed that in cases when the subtrahend is greater than the minuend in more than 20 % of the cases, our proposal outperforms the referenced design (see the vertical dotted line in Figure 7.1).

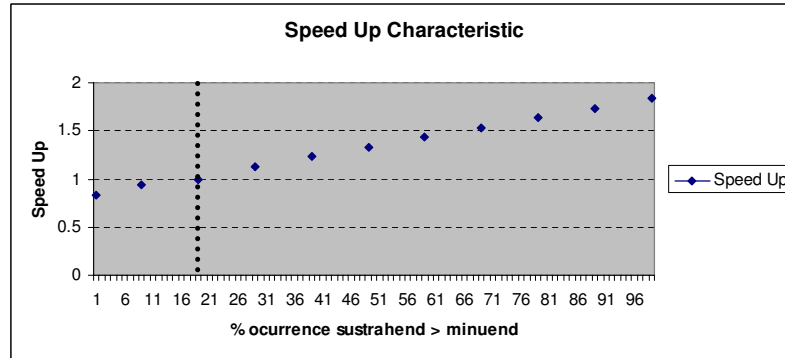


Figure 7.1: Universal Adder - Speed up characteristics

7.2 System Integration of the SAD Unit

In this section, we integrate the SAD arithmetic accelerator into a system with a Microblaze processor. In this way, we can evaluate the impact of our design on the overall performance of the entire system. We have estimated the time required for processing a SAD kernel, implemented as a vector coprocessor, presented in Chapter 6. The achieved results are compared with the pure software execution of the same SAD kernel over a soft-core IP (MicroBlaze).

Setup of the Prototyping System: A general purpose processor was implemented on the Xilinx Virtex II Pro technology. More specifically, we instantiated the Xilinx MicroBlaze, a 32-bits RISC reconfigurable soft-core processor with the following configurations: MicroBlaze processor, UART, 2 Timers, GPIOs, USB, external bus controllers (memory) and the Fast Simplex Link (FSL) channels. The targeted device was an xc2vp30-7ff1696 FPGA. The synthesis report for this SoC gives the following numbers presented in Table 7.3:

Table 7.3: Microblaze based SoC (@ 100 MHz)

Resource Type	Used	Available	HW utilization
Slices	1277	13696	9 %
Slices FF	1501	27392	5 %
LUTS	1928	27392	7 %
BRAM	32	136	23 %

A SAD kernel for motion estimation was coded in C, compiled and executed on the Microblaze SoC architecture described above for an $N=M=4$ SAD macro-block. Two macro-blocks, *Block1*-the reference frame and *Block2*-the searched one, are used as inputs to compute this kernel. The SAD kernel performs subtraction, then computes the absolute value, and finally accumulates the previously obtained results as described in the following pseudo code.

```

1: int MotionEstimation() {
2:   for (i = 0; i < N; i++)
3:     for (j = 0; j < M; j++) {
4:       if ((v = Block1[i][j] - Block2[i][j]) < 0) v = -v;
5:       s+ = v;
6:     }
7:   return s;
8: }
```

The execution time was measured in processor cycles, using the internal timer. We have found that Microblaze requires on average 20 cycles for processing a pel. In particular, the considered SoC requires $3.6 \mu s$ for $N=M=4$.

The SAD Vector Co-Processor: We have implemented the vector-coprocessor presented in Chapter 6 (see Figure 6.4), considering the SAD arithmetic accelerator, presented in Chapter 2. The soft-core vector coprocessor was implemented with the help of Xilinx EKD 8.1i, synthesized and functionally validated using Xilinx ISE 8.1i [135]. The target device was VIRTEX-II PRO xc2vp30-7ff1696 FPGA. Table 7.4 summarizes the results in terms of hardware utilization of the complete micro-architecture of the Vector Co-processor.

Table 7.4: Vector coprocessor: SAD case

Resource Type	Used	Available	HW utilization
Slices	1861	13696	13 %
Slices FF	1125	27392	4 %
LUTS	1248	27392	9 %
DRAM	64	136	47 %

The Communication Interface: The communication link between the MicroBlaze and the SAD coprocessor is established with the FSL point-to-point communication channel. Up to 8 channels are available in the nowadays technology [136]. The interfaces are used to transfer data in 2 clock cycles to and from the processor register file to hardware running on the FPGA device. Figure 7.2 depicts the link between the Coprocessor and the GPP through this dedicated channel. Each channel implements a uni-directional point to point FIFO-based communication. FIFO depths can be as low as 1 and as deep as 8K. The FSL is based in a simple protocol that uses a CLK when the instantiation of the channel is considered in a synchronous FIFO mode. Two reset signals (synchronous and asynchronous) not shown in Figure 7.2 are available for initialization purposes. A 32-bit data bus connects the master and the slave to the shared FIFO memory. A control signal is used to synchronize the data transfers. Separate write and read control signals provide the necessary synchronization for the FIFO memory. Signals indicating a full FIFO are also provided. A complete description of this communication link can be found in [136].

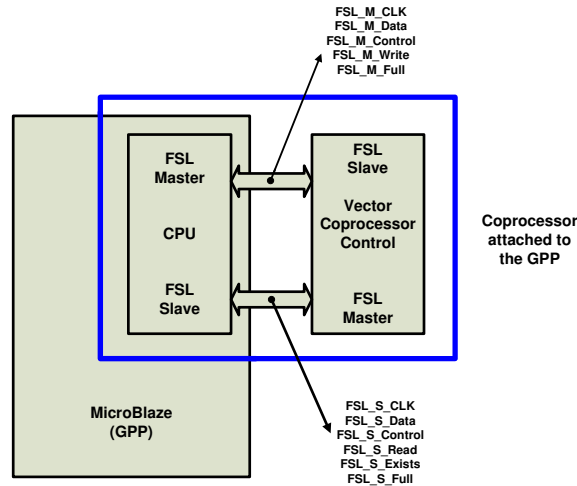


Figure 7.2: GPP and vector coprocessor interface - Fast simplex link dedicated channel

The master and slave of the FSL can work at different rates. In our case, the GPP runs at 100 MHz and the SAD-coprocessor at 40 MHz. When the GPP runs at 100 MHz, 60 ns will be necessary to initialize the register files of the accelerator with the number of elements to process and initialize the addresses for sources and results. The vector coprocessor requires 3 cycles to

fetch, execute and store the result, thus considering a maximum delay of 25 ns presented by the SAD unit, 100 ns will be required to process a macro-block in the proposed arithmetic hardware accelerator. The total amount of time necessary to complete the SAD processing for $N=M=4$ will be 160 ns (speed up: 23 times).

Comparison: The comparison results between the pure software execution and the execution on a SAD hardware enabled system is presented in Table 7.5. Our arithmetic accelerator is able to process in 25 ns 8 pels, which normalized to the Microblaze cycles, is equivalent to 3.2 pels per cycle (speed up: 64 times). The processing of several macro-blocks only will require a single initialization stage, hiding the initial cycles for AGEN set up.

Table 7.5: Software - hardware comparison

	SW = MicroBlaze	HW=MicroBlaze + HW accelerator
Performance	20 cycles per pel	3.2 pels per cycle
% slices used (area)	9 %	13%
Latency	3.6 μ s	160 ns
Speed-up	1	23 - 64

The parallel memory with interleaved data distribution organization, the AGEN unit, and the arithmetic accelerator that collapses 8 absolute value operations and 8 additions allow a speed up of 64 times. Figure 7.3 illustrates the computation considered by the general approach in (a) and by our own approach in (b).

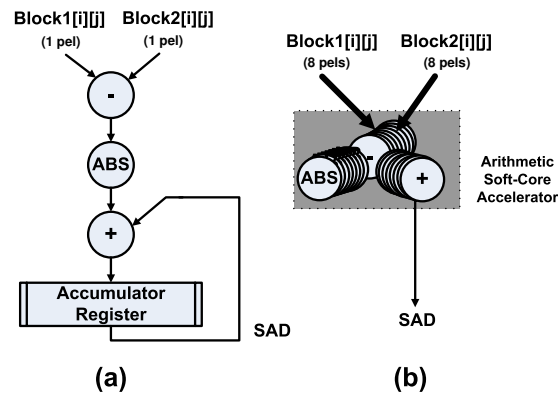


Figure 7.3: SAD processing: (a) Regular approach. (b) Our approach with unit collapsing

A Pipelined Implementation: The arithmetic accelerator for SAD processing proposed in this research has basically three computational steps: 1) Cout computing 2) Partial product reduction ((3:2)counters adder tree) and 3) Final addition. A pipelined version of the arithmetic accelerator proposed in Chapters 2 and 3 can be implemented. The final adder is the bottleneck in this new pipelined structure. In the targeted FPGA technologies, this adder has a 6 ns cycle, which corresponds to an operating frequency of 166.6 MHz. This frequency requirement can be still fulfilled by the proposed AGEN and implemented over the same technology (see Chapter 6), the AGEN generate addresses every 6 ns. Therefore, comparing the 20 cycles per pel required for SAD processing over Microblaze and the possible 13.3 pels per cycle required by the pipelined version of the arithmetic accelerator (running at 166.6 MHz), a speed-up of up to 266 can be achieved.

7.3 Hardware Reutilization

In this section we evaluate the degree of utilization of common hardware by the different designs that we have proposed. The proposed arithmetic units present important degree of reutilization to set-up the multiple operational arithmetic accelerators.

The collapsed units presented in Chapters 2 and 3 are described in Figure 7.4(a) and Figure 7.4(b) schematically.

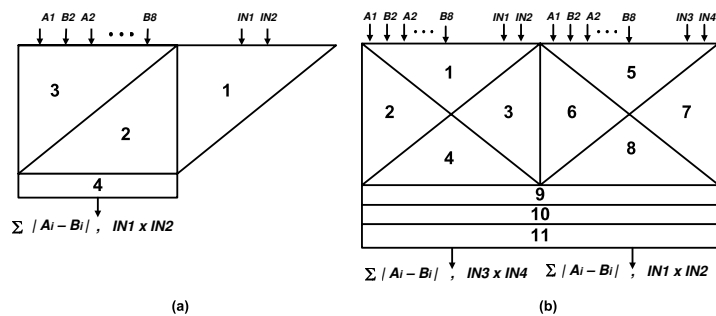


Figure 7.4: Arithmetic soft-cores accelerators: (a) AUSM [5], (b) AUSM extension [6].

Table 7.6 summarizes the amount of hardware reused to construct the various functionalities. These two highly collapsed arithmetic units evidence that it

is possible to share hardware to support different functionalities in a common and single multifunction unit. For example, multiply sub unit have used sections 1, 2, 3, 4, 5, 6, 9 and 11 to build this operation (see Figure 7.4(b)).

Table 7.6: Hardware reuse - AUSM extended

Operation	Sections utilized of the whole array	Reuse of whole array %
Integer Multiply †	1-6, 9, 11	80
Fractional Multiply†	1, 3-8, 9, 11	80
Half 4x4 SAD †	1-4,9, 10, 11	50
Half 4x4 SAD †	5-8,9, 10, 11	50
Integer MAC	1-6, 9, 10, 11	95

† : The units operate with unsigned, signed magnitude and two's complement representation.
The MAC unit operates only with two's complement representation

Furthermore, the arithmetic accelerators presented in Chapters 4 and 5, has been constructed using the same paradigm. Figure 7.5(a) depicts schematically the accelerator presented in Chapter 4 in which the sub-units embedded in this accelerator have the same characteristics as the proposed in Chapters 2 and 3. The reuse of hardware for both functionalities, dense and sparse, reaches the 60 % of whole unit. Regarding the schematic of the universal adder depicted in Figure 7.5(b), the twelve operations ⁱ collapsed in this arithmetic accelerator share almost 40% of the total hardware resources.

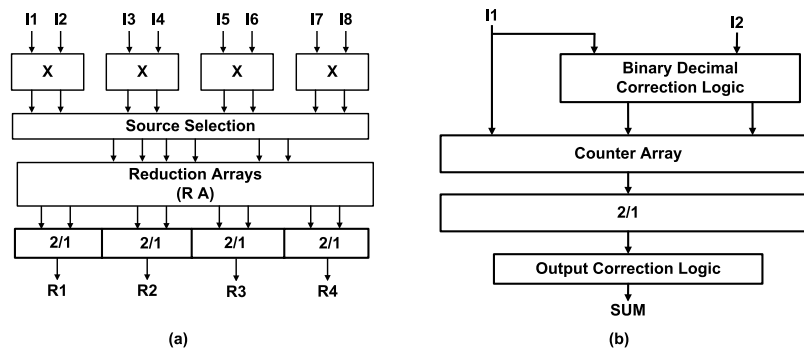


Figure 7.5: Arithmetic accelerators: a) Dense Sparse Matrix Vector Multiply Unit b) Universal Adder

ⁱUniversal addition performs effective addition/subtraction operations on unsigned, sign-magnitude, and various complement representations for binary and BCD representations

A summary of the hardware used in each one of the arithmetic accelerators is presented in Table 7.7. We can see that important blocks of hardware are common for the different arithmetic soft-core accelerators. In addition, Cout logic and muxes are used in all considered designs.

Table 7.7: Common hardware blocks - Arithmetic soft-core accelerators

Operations	[5] AUSM	[6] AUSM extended	[110] Dense/Sparse Matrix-Vector Unit	[3] Universal binary & BCD Adder
2:1 CPA	11	11	11 (4 times)	4
Counter Array	1,2,3,4,5,6	1,2,3,4,5,6,7,8,11	1,2,3,4,5,6,7,8,11	11

From Table 7.7, it can be concluded that some unrelated applications, that require hardware accelerators can share basic blocks such as: array of counters, logic for Cout computing for arithmetic comparisons, multiplexers for routing data and counters. This suggests that there are potential for further inter-application hardware reuse. Adaptable arithmetic accelerators can be instantiated (constructed) on demand. The functions can be configured with multiplexers as suggested in the collapsed units presented in this work. Furthermore, using the run-time reconfiguration (RTR) approach [41, 137–139], the basic units (e.g. multiplier, SAD, BCD adder, etc), which are collapsed into the arithmetic accelerators, can be instantiated “alone” for performing one of the polymorphous behaviors embedded into their design. With this approach, the latency for a particular functionality can be reduced substantially (see, e.g., subsection 4.2.3).

The idea to have a common and basic hardware, such as an *Counter Array* from which different functionalities can be adapted; (e.g., multiplier, SAD and multiple addition accelerators) was the intrinsic idea behind the proposed arithmetic accelerators. With this approach, instead of reconfiguring the entire new functionality, only the differences can be reconfigured; thus diminishing the reconfiguration latency. For example, in Chapter 2, we have reported our experimental results following this approach. Using the technique for two partial reconfiguration flows [61], we found that once the multiplier unit is instantiated, we only need 50 % of the configuration stream data to obtain the SAD unit. Thus, with the use of common blocks, it is possible to diminish the set up time of the functionalities between a multiplier and an 8-input SAD operation. We have to point out that no pre-placement constraints have been used in this

experiment, we rely on the best effort of the compiler. Improved results can be obtained if low level placement tools [140] are used to make partial reconfiguration, assuring that the logic blocks selected are reused to set up the new functionality.

7.3.1 Architectural Adaptability

In Appendix A, we have presented a short survey on some popular reconfigurable architectures. The outcome of that architectural review assessed that adaptable processors present in their data-path units (acceleration units) the following common characteristics:

1. Large interconnection networks for connect the PE arrays and large latency for data communication.
2. Data-path units with dedicated and fixed functional units. The rearrangements are at the arithmetic operators levels (coarse grained units) without any merged execution in their designs (poor reuse of hardware).
3. Unused portions of the data-path when a particular kernel is supported. Some PE are not used in a particular operations.
4. Poor adaptability when customized functionalities, different to the coarse grain arithmetic operators are executed in those data-paths.

Table 7.8 summarizes our findings of the surveyed adaptable processors.

Contrary to the surveyed approaches, the collapsed approach used in the arithmetic accelerators proposed in this research work allows: substantial reuse of hardware, low latencies in the communication networks, and highly customizable, yet easily adaptable functional units. We envision the construction of arithmetic accelerators considering the following:

1. Common and basic logic blocks should be designed to support the functionality of several operand addition related operations. Media applications require operations that deal with multiple data representations like unsigned, sign-magnitude and complement representations.
2. The common blocks can be configured in advance or even created as hard IP, e.g., a set of counters. Therefore, the hardware differences

Table 7.8: Comparison of adaptable processor features

Adaptable Processor	Data-path flexibility	Poor utilization of the FU available	FU interconnection	Poor FU adaptability
VSP	5 - 52 bits	No (few elements)	Crossbar	Fix FU
PRISM	16 bits	Created on-demand some C routines	Shared bus	-
RAW	8 bits	Prototype 4x4	Large network	Fix GPPs
PipeRench	3-bits	Yes (array PEs)	Large Pipeline Network	Flexible FU
Garp	2-bits	Yes (array PEs)	Large resources (2-bit buses)	Flexible FU
MorphoSys	16 bits	Yes (array PEs)	Array of 8x8	Programable ALU
Matrix	8-bits	Yes (array PEs)	Collection of 8 bit buses	Fix ALU
RaPiDs	16 bits	Yes (linear array PEs)	Configurable large network	Flexible multiple resources
Adres	32 bits	Yes (array PEs)	Configurable network	Programable FU
MOLEN	on demand	on demand	-	-

needed for performing a particular operation will be reconfigured partially based on the hardware differences between the common basic array and the new needed functionalities, instead of programming totally the entire desired operation as stated in [16].

3. Large acceleration units can be constructed based on various basic logic blocks that can adapt slightly their functionality to cover the large application requirements. These large acceleration units should be designed to work with different representations and formats, covering with this approach more than a simple application domain.
4. Adaptable fabrics, such as nowadays FPGA devices, should be used to support flexible solutions for the design of the new application specified ISAs for adaptable processor scenarios.

7.4 Conclusions

In this chapter, we presented a comparison of our stand-alone arithmetic units with related works. The occupied silicon area and the unit latency were used as comparisons criteria. We also presented the results of mapping a complete micro-architecture comprising a Microblaze GPP and a SAD coprocessor onto Xilinx Virtex II Pro technology. We have compared the performance of the system augmented with our SAD accelerator with pure software execution of the same kernel. Experimental results indicated speed-up of 64 times. We estimate that this acceleration can potentially reach 266 times speedup with a deeply pipelined version unit of the SAD accelerator. We also evaluated the degree of utilization of the shared hardware in our units. Finally, we provide an architectural investigation and comparison of our approach to existing related work. Comparison results suggest that our proposal allows better adaptability, avoid into the solutions large interconnection networks and at the same time presents the necessary customization to compete with the state of the art when latency is considered.

Chapter 8

General Conclusions

We have considered performance efficient hardware designs of key complex arithmetic operations for general purpose and embedded systems. Our main approach was to collapse several operations into one complex unit exploiting functional commonalities. We have considered reconfigurable hardware arithmetic units that perform specific operations like Multiply (integer and fractional), MAC, SAD, matrix dense/sparse multiply/add and universal addition, including BCD operations. The high data throughput requirements of those arithmetic accelerators have been met by a proposed interleaved memory organization with an address generation unit (AGEN). The AGEN delivers high-bandwidth address generation, allowing fast data retrieval. We have considered the Virtex II Pro technology as a prototyping platform for the proposed designs.

This chapter is organized in three sections. Section 8.1 summarizes the main conclusions we have obtained from the presented research efforts. In Section 8.2, we highlight the main contributions of this dissertation. In Section 8.3 we present a summary of the advantages of the arithmetic accelerators designed with the collapsed hardware approach. Finally, in Section 8.4, we propose some open research directions motivated by short discussions.

8.1 Summary

In Chapter 2, we introduced the design of an adaptable IP that collapses some multi-operand addition operations into a single array. Particularly, we

consider multiply and Sum of Absolute Differences (SAD) operations. The proposed array is capable of performing the aforementioned operations for unsigned, signed magnitude, and two's complement notations. The array was designed around a (3:2)counter array that possess three main operational fields. An 66.6 % of the (3:2)counter array is shared by the implemented operations providing an opportunity to reduce reconfiguration times. The synthesis result for an FPGA device, of the new structure was compared against other multiplier organizations. The obtained results for a Virtex II Pro-6 FPGA technology, indicate that the proposed unit is capable of performing a 16 bit multiplication in 23.9 ns. An 8 pair input SAD can be computed in 29.8 ns. Even though the proposed structure incorporates more operations, the extra delay required over conventional structures is very small. It is in the order of 1% compared to a regular Baugh&Wooley multiplier.

In Chapter 3, the array presented in Chapter 2 is extended with more functionalities while the original ones are preserved. The AUSM extended collapses eight multi-operand addition related operations into a single and common (3:2)counter array. We consider for this unit multiplication in integer and fractional representations, the Sum of Absolute Differences (SAD) in unsigned, signed-magnitude and two's complement notation. Furthermore, the unit also incorporates a Multiply-Accumulation unit (MAC) for two's complement notation. The proposed multiple operation unit was constructed around 10 element arrays using counter reduction techniques. It is estimated that 6/8 of the basic (3:2)counter array structure are shared by the operations. The obtained results of the presented unit indicates that is capable of processing a 4×4 SAD macro-block in 36.3 ns. It takes 30.4 ns to process the other operations using a VIRTEX II PRO xc2vp100-7 FPGA device.

In Chapter 4, we presented a reconfigurable hardware accelerator for the processing of fixed-point-matrix-vector-multiply/add operations. The arithmetic soft-core accelerator unit is capable to work on dense and sparse matrices with several popular formats namely CRS, BBCS and HiSM. The resulting prototype hardware unit accommodates 4 dense or sparse matrix inputs. Due to its parallel design it achieves 4 multiplications and up to 12 additions at 120 MHz when implemented on an xc2vp100-6 FPGA device, which corresponds to a throughput of 1.9 GOPS for both the dense and sparse matrices.

Chapter 5 introduces an adder/subtractor unit that combines Binary and Binary Coded Decimal (BCD) operations. The proposed unit uses effective addition/subtraction operations on unsigned, sign-magnitude, and two's/ten's complement representations. Our design overcomes the limitations of previously reported approaches that produce some of the results in complement representation when operating on sign-magnitude numbers. When reconfigurable technology is considered, our estimation indicates that 40 % of the hardware resources are shared between the different operations. This makes the proposed unit highly suitable for reconfigurable platforms with partial reconfiguration support. The proposed design, was compared with some classical adder organizations after synthesis targeting 4vfx60ff672-12 Xilinx Virtex 4 FPGA. Our design achieves a throughput of 82.6 MOPS with almost equivalent area-time product when compared to other approaches.

In Chapter 6, we describe an efficient data fetch circuitry for retrieving several operands from an 8-bank interleaved memory system in a single machine cycle. The proposed address generation (AGEN) unit operates with a modified version of the low-order-interleaved memory access approach. Our design supports data structures with arbitrary lengths and different (odd) strides. The experimental results shown that the AGEN unit is capable of producing 8 x 32-bit addresses every 6 ns for different stride cases when implemented on VIRTEX-II PRO xc2vp30-7ff1696 FPGA devices using trivial hardware resources.

In Chapter 7, we compared our stand-alone designs with recent related works. Furthermore, we present the results of a complete system integrating the SAD unit as a coprocessor of a Microblaze GPP and present the performance analysis against soft-core and hardcore IP processors.

8.2 Contributions

The main contributions of this dissertation are described in the followings.

- Our major contributions in speeding up media applications through the proposed arithmetic soft-core accelerators are:
 - We have proposed and implemented an Arithmetic Unit for collapsing SAD and Multiplication operations (AUSM). Eight multi-operand addition related operations were collapsed into a sin-

gle and common (3:2)counter array. We proposed a single unit which integrated the multiply operation for integer and fractional representations, the Sum of Absolute Differences (SAD) and the Multiply-Accumulation (MAC) operations. Our evaluations indicate that the proposed SAD unit is 2.3 times faster than related acceleration units. The multiply operation, executed in our unit operates in universal notation and suggest similar latency to Baugh & Wooley Multiply unit implemented over the same technology.

- We have proposed and implemented the Adaptable Fixed Point Dense and Sparse Matrix-Vector Multiply/Add Unit. This arithmetic accelerator is capable to achieve 4 multiplications and up to 12 additions of dense or sparse matrices in a concurrent manner. We designed an algorithm which allocates dynamically the NZE in a collapsed multiply and multiple-addition hardware. The achieved result for these arithmetic accelerator unit indicate that our reconfigurable proposal is competitive to related highly customized hardware accelerators. Although, we did not achieve speed ups compared with the fast dense matrix multiplier units (we have the same performance for sparse matrices), reusing the same computational resources.
 - We have proposed and implemented an Adaptable Universal Adder that supports 12 operations. The proposed unit uses effective addition/subtraction operations on unsigned, sign-magnitude, and various complement representations. Our design overcomes the limitations of previously reported approaches that produce some of the results in complement representation when operating on sign-magnitude numbers. Thus, previous solutions will require at least one cycle more to produce the correct result (assuming zero latency in detection of the case). The latency of our multi-operation unit is 21 % slower compared with the fastest BCD adder considered. Nevertheless, when the occurrence of operations with the subtrahend greater than the minuend is greater than 20 %, the Adaptable Universal Adder is at least two times faster than the fastest related proposal.
- We integrated our units into a complete reconfigurable system :
 - We have proposed an address generation (AGEN) for interleaved memory organization with high parallel computational charac-

teristics. Our design allowed the generation of 8 - 32-bit addresses every 6 ns for different stride cases. When implemented on VIRTEX-II PRO xc2vp30-7ff1696 FPGA device, using trivial hardware resources. This approach reduces the number of cycles required for data transfers between the memory and the arithmetic accelerators providing high bandwidth utilization and short hardware critical paths.

- We have proposed and implemented a vector coprocessor that is composed by an efficient data fetch circuitry for retrieving several operands from a n-bank interleaved memory system in a single machine cycle.
- We integrated our units into a complete reconfigurable system. We have evaluated our proposal for the SAD case. The proposed organization and architecture to deal with the unrolled SAD kernel was evaluated. We have achieved an accelerations of 64 times compared with pure software execution on the MicroBlaze RISC processor over the same technology a VIRTEX-II PRO xc2vp30-7ff1696 FPGA device.

Overall, we have proposed arithmetic soft-core accelerators with universal characteristic able to perform efficiently several operations on the same hardware. In our work, we consider commonly used number notations such as unsigned, sign magnitude, two's/ten's complement representations.

8.3 Advantages of the Collapsed Arithmetic Hardware

Our analysis and experiments suggest that collapsing several arithmetic operations into one complex arithmetic unit can be beneficial in designing performance efficient hardware. We identify the following advantages of the considered function collapsing design approach:

- **Efficient hardware utilization:** High levels of common hardware reutilization among the functions considered allows higher computational densities, e.g., smaller portions of silicon can support larger number of arithmetic operations.
- **Improved performance:** Collapsing multiple operations into a single unit, potentially allows faster data processing due to reduced communi-

cation cost among the different modules compared to traditional multi-module design approaches.

- Applicability for adaptable hardware technologies: Considering contemporary partial reconfigurable hardware, the proposed designs can be easily implemented in run time reconfigurable systems. Providing capabilities to configure the basic functionality first and then to configure a particular functionality in short time, allows to our designs to attain higher overall performance and higher energy efficiency, especially in designs with severely constrained reconfigurable hardware resources.

8.4 Proposed Research Directions

We suggest the following directions to further extend the proposed research.

- Although no experiments for energy evaluation were carried out, the considered collapsing approach is potentially energy efficient. A single unit of high degree of hardware utilization among the supported arithmetic units suggest less power dissipation compared to a collection of autonomous arithmetic units, consuming more energy both for communication and for processing. This aspect should be carefully investigated.
- Addition and multiplication are fundamental operations in current embedded processor data-paths. An interesting research direction could consider the creation of new adaptable technologies, for example new FPGAs with a set of (3:2)counters as hard IP blocks, in the same way that were introduced MUXs, XORs and regular gates like AND and OR in current FPGA devices. The possibility to rearranges the interconnection in a counter array through MUXs has been demonstrated in this research work. This approach allowed the implementation of several several functionalities reusing the same hardware resources. An array of counters with adaptable characteristics can be the starting point in the creation of new and fully customizable arithmetic accelerators with improved performance.
- An exhaustive investigation should be carried out for the creation of an interleaved memory system and AGEN capable to operate with any stride. Thus more investigation should be carried out in order to solve this problem.

- Considering the achieved results in the arithmetic accelerators in this dissertation, a tool for the automatic generation of arithmetic accelerators can be considered as a investigation field. The tool should consider the collapsing approach, combining basic hardware elements, such as counter arrays for supporting several functionalities with different kind of representations. Perhaps some kind of expert systems can help to achieve this goal.
- In this dissertation, we have explored fixed point arithmetic only. An interesting future research direction would be consider the floating point arithmetic operations for collapsing into universal arithmetic units. Floating point sparse matrix multiplication is one promising research direction.

Appendix A

Reconfigurable architectures survey

In this appendix, we provide an overview of some popular adaptable processors proposed by the academia and industry.

This survey compares the main approaches used to design the adaptable processors, analyzing their advantages and disadvantages to respect to the complexity and potential performance of their reconfigurable processing units.

A.0.1 A General-Purpose Programmable Video Signal Processor (VSP)

The creation of an architecture tailored for video signal processing with hard deadline was introduced at the end of the 80s by Philips [7, 141]. The VSP real-time video signal processing was constructed around a group of pipelined processing elements (PE) that communicate with the rest of the system using a crossbar switch. The PE denominated in this proposal “Arithmetic and Logic Elements (ALE)” can be configured to perform operations like Booth multiply, addition, subtraction, comparison and unsigned multiply. The loosely coupled architecture proposed is configured dynamically, achieving some functionalities that are being limited by the concatenation of operations offered by the fixed ALE. We can view the data-path of this proposal as hard adaptable arithmetic accelerator with medium to coarse grained elements. The ALEs can have widths from 10 to 52 bits. The architecture of this configurable ALE is

presented in Figure A.1:

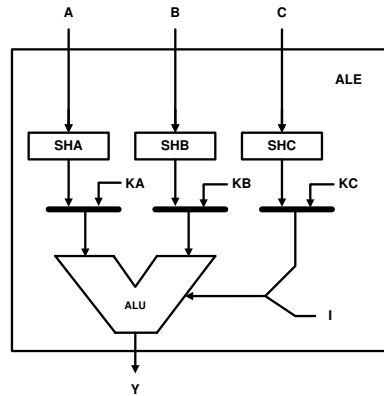


Figure A.1: Contents of the ALE [7]

An ALE consists of three barrel shifters (SHA, SHB and SHC) and an ALU. The inputs A, B and C are connected to the crossbar switch, so that two data operands and one operator instruction (input C) can be fed to the PE. Also a control memory can feed data constants (KA, KB and KC) or instructions directly to the ALE.

A.0.2 Processor Reconfiguration through Instruction-Set Metamorphosis (PRISM)

PRISM [142] is based on the Motorola 68010 processor (core processor) which was augmented with an FPGA board containing four Xilinx 3090 devices in a loosely coupled organization. A shared 16 bit bus is used for the communication between the 68010 processor and FPGA devices. The compiler maps dedicated kernels into the reconfigurable devices on a C-function level basis. The main drawback of this work is the latency between processor and FPGA accelerator communication (large interconnection delay). The second prototype, PRISM-II [8], brought the host processor and FPGAs closer together as can be seen in Figure A.2. The AMD Am29050 is directly connected to three Xilinx 4010's, improving the communication latency and increasing significantly the number of I/O pins resources.

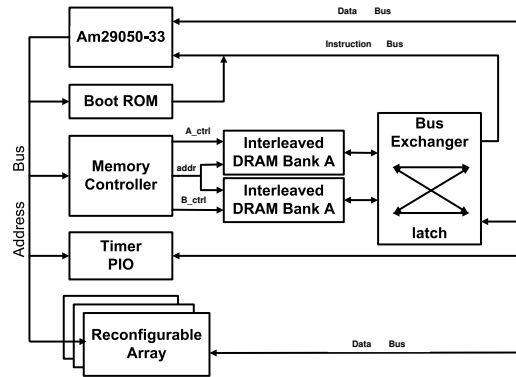


Figure A.2: PRISM II Hardware Platform [8]

A.0.3 Reconfigurable Architecture Workstation (RAW)

RAW [9] was constructed with 16 identical PEs arranged in a programmable tiled organization, where the tiles are connected to its nearest neighbors by the dynamic and static networks (see programmable router in Figure A.3). Each tile has compute resources such as a processor that runs a MIPS ISA style processor with 32 KB SRAM data memory, and a 32 KB SRAM instruction memory [143]. The tiled architecture requires 6 hops for a data value to travel between two non neighboring corners. This type of communication overhead intrinsic of the network on chip approaches [144, 145], is used for hiding the wire delay problem [146, 147] of multiprocessor systems. The RAW coarse grained architecture resembles a multiprocessor system and their adaptability is limited by the number of tiles. Another drawback of RAW is their large configuration time for the interconnect switches (several instructions per switch [9]). This architecture seems adequate for the exploration of the high performance computation (due to their huge granularity in their PEs).

A.0.4 PipeRench

The PipeRench [10] proposal exploits the idea to reconfigure the pipeline on demand (per-application basis). A large logical design (accelerator), is divided and mapped in simpler pieces of hardware, each one in a single stage. The set of pipelined stages (see Figure A.4(a)) denominated “Stripes” are composed by PEs which contain an ALU and two barrel shifters. The ALU consists of a 3-bit LUTs, carry chain support circuitry, zero-detection logic. The connectivity

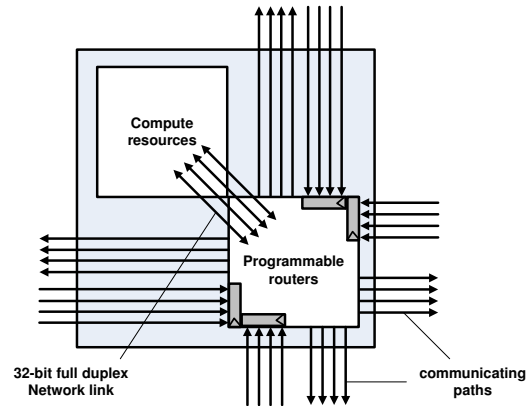


Figure A.3: RAW architecture [9]

of each PE is achieved through a global I/O bus and through the interconnect network. The PEs can retrieve data from register and from another PEs outputs that can be located in the same or another stripe. Figure A.4(b) depicts the architecture of an PE element. This loosely coupled organization attaches the reconfigurable fabric and the host processor through a PCI bus.

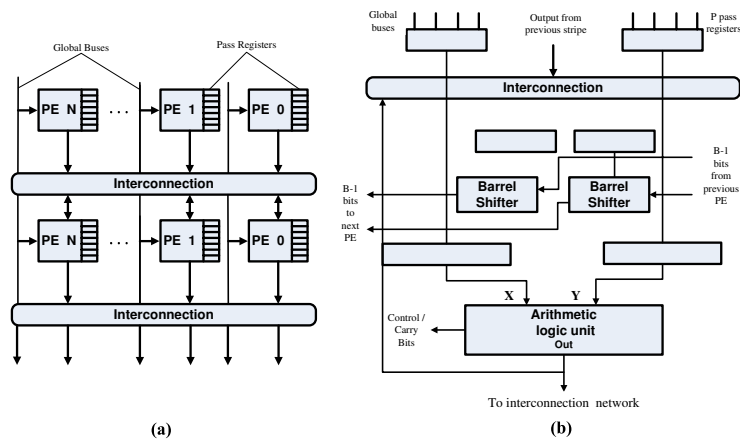


Figure A.4: PipeRench architecture [10]

A.0.5 Garp: a MIPS Processor with a Reconfigurable Coprocessor

This reconfigurable architecture combines a reconfigurable hardware with a standard MIPS processor on the same die in a closely coupling organiza-

tion [11] (see Figure A.5(a)). The reconfigurable array on Garp is composed by entities called blocks, represented by squares in Figure A.5(b). The first block from left to right is intended for controlling tasks, while the rest of the blocks in each row are logic blocks that operate in values of 2 bits. Garp as DISC [148] divides the array in rows for implementing a multi-bit operation across a row. Thus, having 23 logic blocks per row, there is sufficient space in each row for an operation of 32 bits plus the necessary blocks to check overflows, etc. e.g. in Figure A.5(b) the second row is used to map a 32-bit comparator.

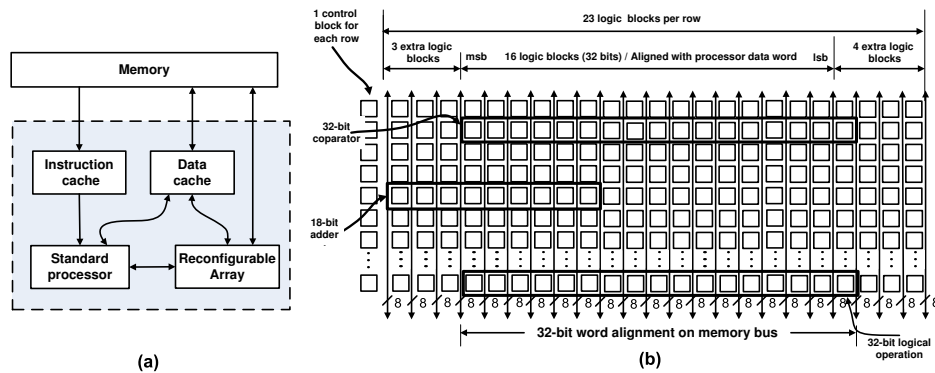


Figure A.5: Garp architecture [11]

Vertical and horizontal wires within the array of logic blocks are used to move data between the blocks. The network wires are grouped in pairs to move two bits. The wire network have several connection patterns optimized for different applications; e.g. horizontal short wires for multi-bit shifts, or (short/long) vertical wires are used to connect arithmetic units.

A.0.6 MorphoSys: a Coarse Grain Reconfigurable Architecture

MorphoSys [12] is an adaptable architecture that combines a TinyRisc general purpose 32-bit RISC processor with a Reconfigurable Cell (RCs) array. Both resources shares a high bandwidth memory interface (DMA controller) as is depicted in Figure A.6 (a). The organization is closely coupled and the reconfigurable array (RC array) acts like a coprocessor.

In the 8x8 RC array, each cell has an ALU/MAC unit and a register file as depicted in Figure A.6 (b). The 8x8 RC array functionality and their interconnection network are configured through the loading of context words which are stored in a context memory in two blocks (one for rows and the other one

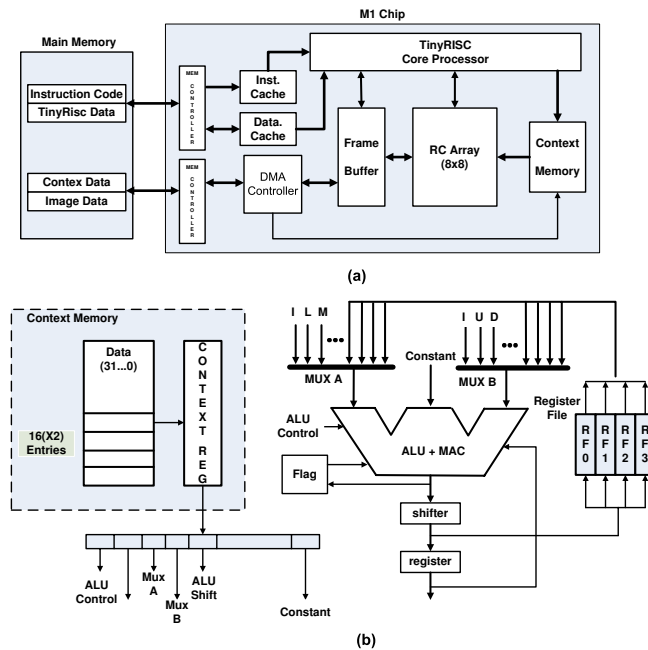


Figure A.6: MorphoSys architecture [12]

for columns). Each block has eight sets of sixteen contexts. The data is provided by two 16-bit input MUXs that select the data operands from different options, presented as MUX A and MUX B in Figure A.6 (b).

A.0.7 Matrix: a Coarse grain Reconfigurable Architecture

Matrix [13] is a coarse grained adaptable computing array. The array was constructed around a set of 8-bit functional units that are used to set up different and adaptable data-paths. Each functional unit consist of a small processor with instruction memory, registers, configuration memory and a control logic. Figure A.7 (a) presents the Basic Functional Unit (BFU) which contains 256x8-bit memory that can also be instantiated as a dual port 128x8-bit memory (register file mode), allowing two reads and one write on each cycle. The 8-bit ALU supports different sets of arithmetic operations and logic functions like NAND, XOR, shift and ADD; when the input is inverted the set augments to AND, OR, and subtraction. The ALU includes also an 8x8 multiply-add-add operation.

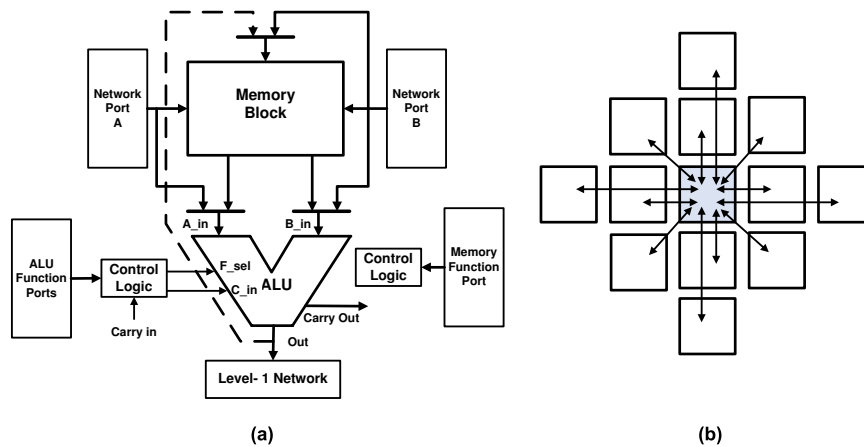


Figure A.7: Matrix architecture [13]

The MATRIX network is constructed hierarchically, based on a collection of 8-bit buses which connect the BFUs. Figure A.7 (b) presents the connection established with the nearest neighbors that requires a single cycle to move data. Also, the MATRIX network is composed by four bypass connections and a global line that connect every row and column. Those connections usually introduce one additional pipeline stage between the producer and the consumer (two different BFUs) for data re-timing purposes.

A.0.8 Architecture Design of Reconfigurable Pipelined Data-paths (RaPiDs)

RaPiDs [14] is a coarse grained field programmable architecture used to construct deep computational pipelines. The authors identify their architecture as a superscalar architecture with hundreds of functional units without cache, many register files and a crossbar interconnect as depicted in Figure A.8. This block diagram presents four main components: data-path, a control-path, an instruction generator and a stream manager. The data-path contains hundreds of functional units including e.g. multi-output Booth-encoded multiplier with a configurable shifter. As mentioned earlier these resources are arranged in a linear array, which can be rearranged to become a linear pipeline. Functional units such as ALUs, multipliers, shifters, and memories are the basic elements of the data-paths. All data-paths as well as the memories are 16 bits wide and those independent units can be chained to perform wider integer operations.

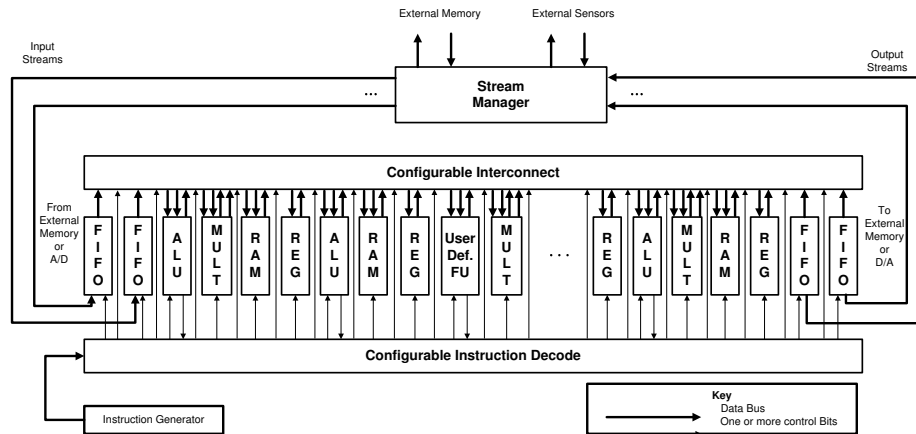


Figure A.8: RaPid architecture [14]

A.0.9 ADRES

The Architecture for Dynamically Reconfigurable Embedded Systems (ADRES), tightly couples a very-long instruction word (VLIW) processor and a coarse grained array called reconfigurable cell (RC), depicted in Figure A.9 (a). The VLIW processor uses a group of FU connected to one multi-port register file. The reconfigurable matrix and the VLIW core share the register files and the FU described above. The three basic elements of the coarse grain adaptable array are: functional units (FUs), storage resources such as register files and memory blocks, and routing resources that include wires, muxes and busses. Each reconfigurable cell has a FU, a register file (RF) and a configuration memory to establish the functionality of the FU (see Figure A.9 (b)). Basically, the computational and storage resources, are connected in a specific topology by the routing resources to form a customized ADRES array. The ADRES architecture is a flexible template that can be freely specified by an XML-based architecture specification language as an arbitrary combination of those RC elements. Furthermore, the ADRES compiler denominated Dynamically Reconfigurable Embedded System Compiler (DRESC) framework assures that applications written in C can be easily mapped onto VLIW and array mode [15, 149]. The reconfigurable matrix is capable to execute word (32 bits) or sub-word level operations.

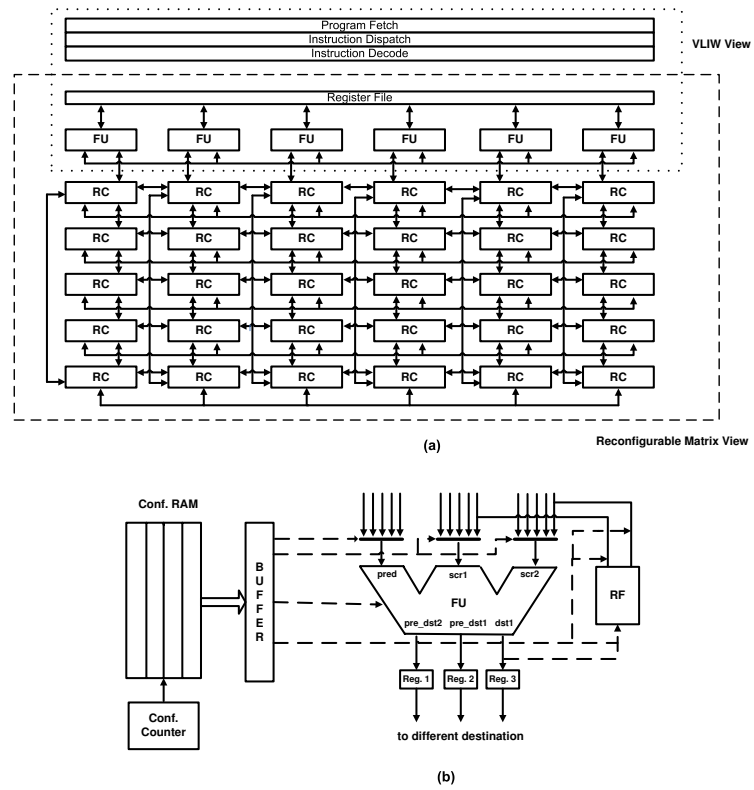


Figure A.9: Adres architecture [15]

A.0.10 MOLEN: a customizable adaptable architecture

The MOLEN $\rho\mu$ -coded processor introduced in [150] and with its organization, ISA and programming paradigm described in [16], merges a GPP and an adaptable coprocessor in a closely coupled organization, Figure A.10 depict the MOLEN machine organization. The Reconfigurable Unit (adaptable processor) consists of a Custom Computing Unit (CCU) and a $\rho\mu$ -code unit. This adaptable processor uses the micro code technique introduced in [151], where the $\rho\mu$ -code is used for carrying out the configuration process of the augmented CCU, as well for the emulation of the execution of the core processing unit and the control over the execution of the CCUs.

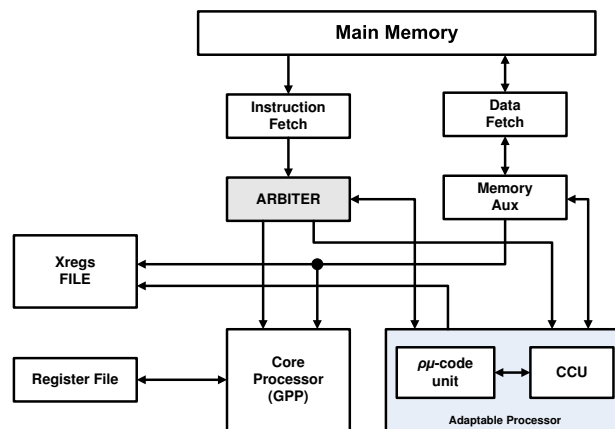


Figure A.10: MOLEN machine organization [16]

The ARBITER [152] (see FigureA.10) partially decodes the fetched instruction, determining where will be issued for execution; it has two alternatives the GPP and the CCU. The envisioned support of operations by MOLEN is divided into two phases: *set* and *execute*. In set phase the CCU is configured and in execute phase the execution of the operations are performed. There are basically three distinctive π ISA (Polymorphic ISA): the minimal, the preferred and the complete π ISA. The **minimal** π ISA provide the working scenario with 4 basic instructions, set, execute and move instructions for interchanging of data between GPP and the XREGs (*movfx* and *movtx*). The **preferred** π ISA includes pre-fetch instructions (*set pre-fetch* and *execute pre-fetch*) for hide the latency reconfiguration, and the **complete** π ISA introduces the *break* instruction for synchronization the execution of parallel instructions.

Bibliography

- [1] I. S. Hwang, “High-Speed Binary and Decimal Arithmetic Logic Unit,” *American Telephone and Telegraph Company, AT&T Bell Laboratories, US patent 4866656*, pp. 1 – 11, Sep 1989.
- [2] H. Fischer and W. Rohsaint, “Circuit Arrangement for Adding or Subtracting Operands Coded in BCD-Code or Binary-Code,” *Siemens Aktiengesellschaft, US patent 5146423*, pp. 1 – 9, Sep. 1992.
- [3] H. Calderón, G. Gaydadjiev, and S. Vassiliadis, “Reconfigurable Universal Adder,” *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 07)*, pp. 186 – 191, Jul. 2007.
- [4] H. Calderón, C. Galuzzi, G. Gaydadjiev, and S. Vassiliadis, “High-Bandwidth Address Generation Unit,” *International Symposium on Systems, Architectures, MOdeling and Simulation (SAMOS VII Workshop)*, pp. 263 – 274, Jul. 2007.
- [5] H. Calderón and S. Vassiliadis, “Reconfigurable Universal SAD-Multiplier Array,” *Proceedings of Computing Frontiers 2005, ACM*, pp. 72 – 76, May 2004.
- [6] H. Calderón and S. Vassiliadis, “Reconfigurable Multiple Operation Array,” *Proceedings of the Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS’05)*, pp. 22 – 31, July 2005.
- [7] R. Mehtani, K. Baker, C. M. Huizer, J. P. Hynes, and J. V. Beers, “Macro-Testability and the VSP,” *Proceedings of the International Test Conference*, pp. 739 – 748, Sep. 1990.
- [8] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh, “PRISM-II Compiler and Architecture,”

- Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 9 – 16, Apr. 1993.
- [9] M. Taylor, J. Kim, J. Miller, F. Ghodrat, B. Greenwald, P. Johnson, W. Lee, A. Ma, N. Shnidman, V. Strumpfen, D. Wentzlaff, M. Frank, S. Amarasinghe, and A. Agarwal, “The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs,” *IEEE MICRO*, pp. 25 – 35, Apr. 2002.
- [10] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, “PipeRench: a Reconfigurable Architecture and Compiler,” *IEEE Computer*, pp. 70 – 77, Apr. 2000.
- [11] J. R. Hauser and J. Wawrzynek, “Garp: a MIPS Processor with a Reconfigurable Coprocessor,” *Proceedings of The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines, 1997*, pp. 12 – 21, Apr. 1997.
- [12] H. Singh, M. H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho, “MorphoSys: an Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications,” *IEEE Transactions on Computers*, pp. 465 – 481, May. 2000.
- [13] E. Mirsky and A. DeHon, “MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources,” *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 157 – 166, Apr. 1996.
- [14] C. Ebeling, D. C. Cronquist, and P. Franklin, “RaPiD Reconfigurable Pipelined Datapath,” *6th International Workshop on Field-Programmable Logic and Applications Field-Programmable Logic: Smart Applications, New Paradigms, and Compilers*, pp. 126 – 135, 1996.
- [15] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, “ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix,” *Proceedings of the 13th International Conference, FPL*, pp. 61 – 70, Sep. 2003.
- [16] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte, “The MOLEN Polymorphic Processor,” *IEEE Transactions on Computers*, pp. 1363 – 1375, Nov. 2004.

- [17] S. Benford, C. Magerkurth, and P. Ljungstrand, "Bridging the Physical and Digital in Pervasive Gaming," *Communications of the ACM*, pp. 54 – 57, Mar. 2005.
- [18] C. Magerkurth, A. D. Cheok, R. L. Mandryk, and T. Nilsen, "Pervasive Games: Bringing Computer Entertainment Back to the Real World," *ACM Computers in Entertainment*, pp. 1 – 19, Jul. 2005.
- [19] European Commission Directorate General II Economic and Financial Affairs: Economy of the Euro zone and the Union, "The introduction of the EURO and the Rounding of Currency Amounts," *II/28/99-EN Euro Papers*, pp. DGII/C-4-SP(99), Mar. 1998.
- [20] M. Cowlshaw, "General Decinal Arithmetic Specification," <http://www2.hursley.ibm.com/decimal/decarith.pdf>, pp. 1 – 19, Mar. 2007.
- [21] C. Wei and B. Kolko, "Studying Mobile Phone Use in Context: Cultural, Political, and Economic Dimensions of Mobile Phone Use," *Proceedings of the IEEE International Professional Communication Conference*, pp. 205 – 212, Jul. 2005.
- [22] I. S. Burnett, F. Pereira, R. V. de Walle, and R. Koenen, "The MPEG-21 Book," *Wiley & Sons*, vol. ISBN: 978-0-470-01011-2, March 2006.
- [23] F. DeKeukelaere, D. DeSchrijver, S. DeZutter, and R. V. de Walle, "MPEG-21 Session Mobility for heterogeneous Devices," *IEEE Transactions on Computers*, vol. 53, pp. 1363 – 1375, Nov. 2004.
- [24] J. Parkhurst, J. Darringer, and B. Grundmann, "From Single Core to Multi-Core: Preparing for a New Exponential," *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design IC-CAD '06*, pp. 67 – 72, Nov. 2006.
- [25] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao, "Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures," *Proceedings of the 34th annual international symposium on Computer architecture ISCA '07*, pp. 35 – 45, Jun. 2007.
- [26] O. Lempel, A. Peleg, and U. Weiser, "Intel's MMX Technology-a New Instruction Set Extension," *Proceedings of the IEEE Compcon '97*, pp. 255 – 259, Feb. 1997.

- [27] S. Thakkur and T. Huff, "Internet Streaming SIMD Extensions," *IEEE Computer*, pp. 26 – 34, Dic. 1999.
- [28] Y. W. Huang, C. Y. Chen, C. H. Tsai, C. F. Shen, and L. G. Chen, "Survey on Block Matching Motion Estimation Algorithms and Architectures with New Results," *Journal of VLSI Signal Processing* 42, p. 297 – 320, Feb. 2006.
- [29] H. F. Ates and Y. Altunbasak, "SAD Reuse in Hierarchical Motion Estimation for the H.264 Encoder," *IEEE International Conference on Acoustics, Speech, and Signal Processing, (ICASSP '05)*, pp. 905 – 908, Mar. 2005.
- [30] A. Janecek and H. Janecek, "Programming Interactive Real-Time Games over WLAN for Pocket PCs with J2ME and .NET CF," *4th Workshop on Networks and System Support for Games, Net Games'05*, pp. 1 – 8, Oct. 2005.
- [31] S. Kotani, A. Inoue, T. Imamura, and S. Hasuo, "A 1 GOPs 8 b Josephson digital signal processor," *Proceedings of International Conference on Solid-State Circuits, 37th ISSCC*, pp. 148 – 150, Feb. 1990.
- [32] G. Panneerselvam and B. Nowrouzian, "Multiply-Add Fused RISC Architectures for DSP Applications," *Proceedings of the IEEE Pacific Rim Conference in Communications, Computers and Signal Processing*, pp. 108 – 111, May. 1993.
- [33] M. F. Cowlishaw, "General Decimal Arithmetic," *IBM Corporation*, Available at: <http://www2.hursley.ibm.com/decimal/>, 2004.
- [34] H. Thapliyal and S. K. Gupta, "Design of Novel Reversible Carry Look-Ahead BCD Subtractor," *Proceedings of the 9th International Conference on Information Technology, ICIT '06.*, pp. 253 – 258, Dec. 2006.
- [35] M. Horowitz, R. Ho, and K. Mai, "The Future of Wires," *Proceedings of the IEEE*, pp. 490 – 504, Apr. 2001.
- [36] E. E. Swartzlander, "Merged Arithmetic," *IEEE Transactions on Computers*, pp. 946 – 950, Oct. 1980.
- [37] S. M. Lee, J. H. Chung, H. S. Yoon, and M. O. Lee, "High Speed and Ultra-Low Power 16X16 MAC Design using TG techniques for Web-based Multimedia System," *Proceedings of the IEEE International Con-*

- ference on Asia and South Pacific Design Automation(ASP-DAC'00)*, pp. 17 – 18, Jan. 2000.
- [38] S. Vassiliadis, B. Blaner, and R. J. Eickermeyer, “SCISM: A Scalable Compound Instruction Set Machine,” *IBM Journal of Research and Development*, pp. 59 – 78, January 1994.
- [39] E. A. Hakkennes, S. Vassiliadis, and S. D. Cotofana, “Fast Computation of Compound Expressions in Two’s Complement Notation,” *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modeling and Applied Mathematics*, vol. 53, pp. 689 – 694, Aug. 1997.
- [40] J. Phillips and S. Vassiliadis, “High-Performance 3-1 Interlock Collapsing ALU’s,” *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modeling and Applied Mathematics*, vol. 43, pp. 257 – 268, Mar. 1994.
- [41] K. Compton and S. Hauck, “Reconfigurable Computing: a Survey of Systems and Software,” *ACM Computing Surveys (CSUR)*, pp. 171 – 210, Jun. 2002.
- [42] S. Vassiliadis, E. Schwarz, and M. Putrino, “Quasi-Universal VLSI Multiplier with Signed Digit Arithmetic,” *Proceedings of the 1987 IEEE Southern Tier Technical Conference*, pp. 1 – 10, Apr. 1987.
- [43] P. Pirsch, N. Demassieux, and W. Gehrke, “VLSI Architectures for Video Compression a Survey,” *IEEE Proceedings*, pp. 220 – 246, Feb. 1995.
- [44] D. Guevorkian, A. Launiainen, P. Liuha, and V. Lappalainen, “Architectures for the Sum of Absolute Differences Operation,” *IEEE Workshop on Signal Processing Systems (SIPS'02)*, Oct. 2002.
- [45] P. Kuhn, “Fast MPEG-4 Motion Estimation: Processor Based and Flexible VLSI Implementations,” *Journal of VLSI Signal Processing*, pp. 67 – 92, 1999.
- [46] S. Vassiliadis, E. Hakkennes, S. Wong, and G. Pechanek, “The Sum-Absolute-Difference Motion Estimation Accelerator,” *Proceedings of Euromicro Conference, 24th*, pp. 559 – 566, Aug. 1998.
- [47] H. Ling, “High-Speed Binary Adder,” *IBM Journal Research and Development*, pp. 156 – 166, Vol. 25 1981.

- [48] O. J. Bebrij, "Carry Select Adder," *IRE Transactions on Electronic Computers*, pp. 340 – 346, Vol. 39 1962.
- [49] M. Lehman and N. Burla, "Skip Techniques for High-Speed Carry Propagation in Binary Arithmetic Units," *IRE Transactions on Electronic Computers*, pp. 691 – 698, Dec. 1961.
- [50] B. Parhami, "Computer Arithmetic: Algorithms and Hardware Designs," *Oxford University Press*, p. 1 – 490, 2000.
- [51] J. Tyler, J. Lent, A. Mather, and N. Huy, "AltiVec™: bringing vector technology to the PowerPC™ processor family," *IEEE International Conference in Performance, Computing and Communications, IPCCC '99.*, pp. 437 – 444, Feb. 1999.
- [52] M. Tremblay, J. O'Connor, V. Narayanan, and L. He, "VIS Speeds New Media Processing," *IEEE MICRO*, pp. 10 – 20, Aug. 1996.
- [53] S. Thakkur and T. Huff, "Internet Streaming SIMD Extensions," *IEEE Computer*, pp. 26 – 34, Dec. 1999.
- [54] S. Vassiliadis, D. S. Lemon, and M. Putrino, "S/370 Sign-Magnitude Floating-Point Adder," *IEEE Journal of Solid-State Circuits*, pp. 1062 – 1070, Aug. 1989.
- [55] I. Amer, C. Rahman, T. Mohamed, M. Sayed, and W. Badawy, "A Hardware-Accelerated Framework with IP-Blocks for Application in MPEG-4," pp. 211 – 214, Jul. 2005.
- [56] J. Olivares, J. Hormigo, J. Villalba, I. Benavides, and E. L. Zapata, "SAD computation Based on Online Arithmetic for Motion Estimation," *Microprocessors and Microsystems*, pp. 250 – 258, Dic. 2005.
- [57] Xilinx, "The XILINX Software Manuals, XILINX 5.2i," http://www.xilinx.com/support/sw_manuals/xilinx5/index.htm, 2003.
- [58] ModelSim PE, "http://www.model.com/products/products_pe.asp,"
- [59] C. R. Baugh and B. A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," *IEEE, Transactions on Computers*, pp. 1045 – 1047, Dic. 1973.
- [60] Xilinx, "Virtex II Pro Platform FPGA Handbook," Oct. 2002.

- [61] I. XILINX, “Two Flows for Partial Reconfiguration: Module Based or Difference Based,” <http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf>, pp. 1 – 28, Sep. 2004.
- [62] N. Yaday, M. Schulte, and J. Glossner, “Parallel Saturating Fractional Arithmetic Units,” *Proceedings of the Ninth great lakes Symposium on VLSI*, Mar. 1999.
- [63] C. S. Wallace, “A Suggestion for Fast Multiplier,” *IEEE Transactions on Electronic Computers*, vol. EC-13, pp. pages 14 – 17, February 1964.
- [64] L. Zhuo and V. K. Prasanna, “Sparse Matrix-Vector Multiplication on FPGAs,” *ACM/SIGDA Thirteenth International Symposium on Field Programmable gate Arrays (FPGA 2005)*, pp. 63 – 74, Feb. 2005.
- [65] P. Stathis, S. Vassiliadis, and S. D. Cotofana, “Hierarchical Sparse Matrix Storage Format for Vector Processors,” *In Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, p. 61a, Apr. 2003.
- [66] Y. Dou, S. Vassiliadis, G. Kuzmanov, and G. N. Gaydadjiev, “64-bit Floating-Point FPGA Matrix Multiplication,” *ACM/SIGDA Thirteenth International Symposium on Field Programmable Gate Arrays (FPGA 2005)*, vol. 15, pp. 86 – 95, Feb. 2005.
- [67] B. C. Lee, R. Vuduc, J. W. Demmel, and K. A. Yelick, “Performance Models for Evaluation and Automatic Tuning of Symmetric Sparse Matrix-Vector Multiply,” *Proceedings of the 2004 International Conference on Parallel Processing (ICPP’04)*, pp. 169 – 176, Jun. 2004.
- [68] S. Vassiliadis, S. D. Cotofana, and P. Stathis, “Block Based Compression Storage Expected Performance,” *In Proceedings of the 14th International Conference on High Performance Computing Systems and Applications (HPC 2000)*, pp. 389 – 406, Jun. 2000.
- [69] E. Roesler and B. Nelson, “Novel Optimizations for Hardware Floating-Point Units in a Modern FPGA Architecture,” *In Proceedings of the 12th International Workshop on Field Programmable Logic and Application (FPL 2002)*, pp. 637 – 646, Aug. 2002.

- [70] M. deLorimeier and A. DeHon, "Floating-Point Sparse Matrix-Vector Multiply for FPGAs," *ACM/SIGDA Thirteenth International Symposium on Field Programmable Gate Arrays (FPGA 2005)*, pp. 75 – 85, Feb. 2005.
- [71] L. Zhuo and V. K. Prasanna, "Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on FPGAs," *In Proceedings on the 18th International parallel and Distributed Processing Symposium (IPDPS 2004)*, pp. 94 – 103, Apr. 2004.
- [72] S. Vassiliadis, S. D. Cotofana, and P. Stathis, "BBCS Based Sparse Matrix-Vector Multiplication: Initial Evaluation," *In Proceedings of the 16th IMACS World Congress on Scientific Computation*, pp. 1 – 6, August 2000.
- [73] J. Choi, "A Fast Scalable Universal Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers," *In The 11th IEEE International Parallel Processing Symposium (IPPS 97)*, pp. 310 – 314, Apr. 1997.
- [74] K. D. Underwood, "FPGA vs. CPUs: Trends in Peak Floating-Point Performance," *In Proceedings of the ACM International Symposium on Field Programmable Gate Arrays (FPGA 2004)*, pp. 171 – 180, Feb. 2004.
- [75] K. D. Underwood and K. S. Hemmert, "Closing the Gap: CPU and FPGA Trends in Sustainable Floating Point Blas Performance," *In Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM 2004)*, pp. 219 – 228, Apr. 2004.
- [76] R. W. Vuduc and H. J. Moon, "Fast Sparse Matrix-Vector Multiplication by Exploiting Variable Block Structure," *Proceedings of the International Conference on High Performance Computing and Communications (HPCC)*, pp. 807 – 816, Sep. 2005.
- [77] L. Zhuo and V. K. Prasanna, "Sparse Matrix-Vector Multiplication on FPGAs," *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, pp. 63 – 74, Feb. 2005.
- [78] I. XILINX, "The XILINX Software Manuals, XILINX 6.1i," http://www.xilinx.com/support/sw_manuals/xilinx6/, 2004.

- [79] European Commission Directorate General, “http://ec.europa.eu/economy_finance/publications,” *European Commission Directorate General II Economic and Financial Affairs*, Mar. 1998.
- [80] S. Microsystems, “BigDecimal (Java 2 Platform SE v1.4.0)” <http://java.sun.com/products>,” *Sun Microsystems Inc.*, 2002.
- [81] M. Palmer, “A Comparison of 8-bit microcontrollers,” *Application note 520, Microchip Technology Inc.*, pp. 1 – 11, Jan. 1997.
- [82] H. Goldstine and A. Goldstine, “The Electronic Numerical Integrator and Computer (ENIAC),” *Mathematical Tables and Other Aids to Computation*, pp. 97 – 110, Jul. 1946.
- [83] C. J. Bashe, W. Buchholz, and N. Rochester, “The IBM type 702: An electronic Data Processing Machine for Business,” *Journal of the Association for Computing Machinery*, pp. 149 – 169, Oct. 1954.
- [84] M. F. Cowlishaw, “Decimal Floating-Point: Algorithm for Computers,” *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH’03)*, pp. 1 – 8, Aug. 2003.
- [85] S. Oberman, G. Favor, and F. Weber, “AMD 3DNow! Technology: Architecture and Implementations,” *IEEE MICRO*, pp. 37 – 48, Apr. 1999.
- [86] M. S. Schmookler and A. W. Weinderger, “Decimal Adder for Directly Implementing BCD Addition Utilizing Logic Circuitry,” *International Business Machines Corporation, US patent 3629565*, pp. 1 – 19, Dic. 1971.
- [87] M. J. Adiletta and V. C. Lamere, “BCD Adder Circuit,” *Digital Equipment Corporation, US patent 4805131*, pp. 1 – 18, Jul. 1989.
- [88] S. R. Levine, S. Singh, and A. Weinberger, “Integrated Binary-BCD Look-Ahead Adder,” *International Business Machines Corporation, US patent 4118786*, pp. 1 – 13, Oct. 1978.
- [89] J. L. Anderson, “Binary or BCD Adder with Precorrected Result,” *Motorola, Inc., US patent 4172288*, pp. 1 – 8, Oct. 1979.
- [90] U. Grupe, “Decimal Adder,” *Vereinigte Flugtechnische Werke-Fokker gmbH, US patent 3935438*, pp. 1 – 11, Jan. 1976.

- [91] L. P. Flora, "Fast BCD/Binary Adder," *Unisys Corp., US patent 5007010*, pp. 1 – 16, Apr 1991.
- [92] M. A. Check and T. J. Slegel, "Custom S/390 G5 and G6 Microprocessors," *IBM J. Res. & Dev.* 43, No. 5/6, pp. 671 – 680, Sep. 1999.
- [93] E. M. Schwarz, M. A. Check, C. L. K. Shum, T. Koehler, S. B. Swaney, J. D. MacDougall, and C. A. Krygowski, "The Microarchitecture of the IBM eServer z900 Processor," *IBM J. Res. & Dev.* 46, No 4/5, pp. 381 – 395, Jul. 2002.
- [94] F. Y. Busaba, C. A. Krygowsky, W. H. Li, E. M. Schwarz, and S. R. Carrough, "The IBM z900 Decimal Arithmetic Unit," *Conference Record of the 35th Asilomar conference on Signals, Systems and Computers.*, pp. 1353 – 1339, Sep. 2001.
- [95] W. Haller, U. Krauch, and H. Wetter, "Combined Binary/Decimal Adder Unit," *International Business Machines Corporation, US patent 5928319*, pp. 1 – 9, Jul 1999.
- [96] W. Haller, W. H. Li, M. R. Kelly, and H. Wetter, "Highly Parallel Structure for Fast Cycle Binary and Decimal Adder Unit," *International Business Machines Corporation, US patent 2006/0031289*, pp. 1 – 8, Feb 2006.
- [97] H. Calderón, G. Gaydadjiev, and S. Vassiliadis, "FPGA based implementation of Reconfigurable Universal Adders," *Technical Report CE-TR-2007-01*, pp. 1 – 21, Jan. 2007.
- [98] J. Corbal, R. Espasa, and M. Valero, "Three-Dimensional Memory Vectorization for High Bandwidth Media Memory Systems," *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35)*, pp. 149 – 160, Nov. 2002.
- [99] R. Espasa and M. Valero, "Exploiting instruction- and data-level parallelism," *IEEE Micro*, pp. 20 – 27, Sep. 1997.
- [100] S. Mamidi, E. R. Blem, M. J. Schulte, J. Glossner, D. Iancu, A. Iancu, M. Moudgill, and S. Jinturkar, "Instruction Set Extensions for Software Defined Radio on a Multithreaded Processor," *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 266 – 273, Sep. 2005.

- [101] S. B. Wijeratne, N. Siddaiah, S. K. Mathew, M. A. Anders, R. K. Krishnamurthy, J. Anderson, M. Ernest, and M. Nardin, "A 9-GHz 65-nm Intel® Pentium 4 Processor Integer Execution Unit," *IEEE Journal of Solid-State Circuits*, pp. 26 – 37, Jan. 2007.
- [102] S. Mathew, M. Anders, R. K. Krishnamurthy, and S. Borkar, "A 4-GHz 130-nm Address Generation Unit with 32-bit Sparse-Tree Adder Core," *IEEE Journal of Solid-State Circuits*, pp. 689 – 695, May. 2003.
- [103] J. Y. Kim and M. H. Sunwoo, "Design of Address Generation Unit for Audio DSP," *Proceedings of 2004 International Symposium on Intelligent Signal Processing and Communication Systems, 2004. ISPACS 2004*, pp. 616 – 619, Nov. 2004.
- [104] J. Cho, H. Chang, and W. Sung, "An FPGA based SIMD processor with a vector memory unit," *Proceedings of the 2006 IEEE International Symposium on Circuits and Systems, 2006. ISCAS 2006*, pp. 525 – 528, May. 2006.
- [105] K. Hirano, T. Ono, H. Kurino, and M. Koyanagi, "A New Multiport Memory for High Performance Parallel Processor System with Shared Memory," *Proceedings of the Design Automation Conference ASP-DAC '98*, pp. 333 – 334, Feb. 1998.
- [106] A. Postula, S. Chen, L. Jozwiak, and D. Abramson, "Automated Synthesis of Interleaved Memory Systems for Custom Computing Machines," *Proceedings of the 24th Euromicro Conference*, pp. 115 – 122, August 1998.
- [107] G. S. Sohi, "High-bandwidth Interleaved Memories for Vector Processors - a Simulation Study," *IEEE Transactions on Computers*, pp. 34 – 44, Jan. 1993.
- [108] K. Hwang and F. A. Briggs, "Computer Architecture and Parallel Processing," *McGraw-Hill*, 1984.
- [109] A. Sez nec and J. Lenfant, "Interleaved Parallel Schemes," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1329 – 1334, December 1994.
- [110] H. Calderón and S. Vassiliadis, "Reconfigurable Fixed Point Dense and Sparse Matrix-Vector Multiply/Add Unit," *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 06)*, pp. 311–316, Sep. 2006.

- [111] I. XILINX, “<http://www.xilinx.com/ipcenter/>,” 2007.
- [112] XILINX-LogiCore, “Dual-Port Block Memory v7.0 - Product Specification,” *DS235 Xilinx*, Dec. 2003.
- [113] M. Sanu, A. Mark, K. Ram, and B. Shekhar, “A 4GHz 130nm Address Generation Unit with 32-bit sparse-tree adder core,” *In The 11th IEEE International Parallel Processing Symposium (IPPS 97)*, pp. 310 – 314, Apr. 1997.
- [114] B. Juurlink, D. Cheresiz, S. Vassiliadis, and H. A. G. Wijshoff, “Implementation and Evaluation of the Complex Streamed Instruction Set,” *Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pp. 73 – 82, Sep. 2001.
- [115] I. XILINX, “The XILINX Software Manuals, XILINX 7.1i,” http://www.xilinx.com/support/sw_manuals/xilinx7/, 2005.
- [116] XILINX-LogiCore, “Adder/Subtractor v7.0 - Product Specification,” *DS214 Xilinx*, Dec. 2003.
- [117] I. XILINX, “http://www.xilinx.com/products/design_resources/mem_corner/,” *Memory Solutions*, 2007.
- [118] C. Wei and M. Gang, “Novel SAD Computing Hardware Architecture for Variable-Size Block Motion Estimation and Its Implementation with FPGA,” *Proceedings of the 5th International Conference on ASIC*, pp. 950 – 953, Oct. 2003.
- [119] M. Sayed, T. Mohamed, and W. Badawy, “Motion Estimation Architecture for MPEG-4 Part 9: Reference Hardware Description,” *International Conference on Electrical, Electronic and Computer Engineering, ICEEC '04*, pp. 403 – 406, Sep. 2004.
- [120] H. Loukil, F. Ghazzi, A. Samet, M. A. BenAyed, and N. Masmoudi, “Hardware implementation of block matching algorithm with FPGA technology,” *Proceedings of The 16th International Conference on Microelectronics, ICM 2004*, pp. 542 – 546, Dec. 2004.
- [121] M. Mohammadzadeh, M. Eshghi, and M. M. Azadfar, “An optimized systolic array architecture for full search block matching algorithm and its implementation on FPGA chips,” *Proceedings of The 3rd International IEEE-NEWCAS Conference*, pp. 327 – 330, Jun. 2005.

- [122] ALTERA, "Device Comparison," http://www.altera.com/cgi-bin/device_compare.pl, Jul. 2007.
- [123] J. W. Jang, S. Choi, and V. Prasanna, "Area and Time Efficient Implementations of Matrix Multiplication on FPGAs," *The First IEEE International Conference on Field Programmable Technology (FPT)*, pp. 93 – 100, Dic. 2002.
- [124] O. Mencer, M. Morf, and M. J. Flynn, "PAM-Blox: High Performance FPGA Design for Adaptive Computing," *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 167 – 174, Apr. 1998.
- [125] A. Amira, A. Bouridane, and P. Milligan, "Accelerating Matrix Product on Reconfigurable Hardware for Signal Processing," *11th International Conference on Field-Programmable Logic and Applications FPL 2001*, pp. 101 – 111, Aug. 2001.
- [126] V. K. Prasanna and Y. C. Tsai, "On Synthesizing Optimal Family of Linear Systolic Arrays for Matrix Multiplication," *IEEE Transactions on Computers*, pp. 770 – 774, Jun. 1991.
- [127] H. Li and M. Sheng, "Sparse Matrix Vector Multiplication on Polymorphic-Torus," *Proceeding of the 2nd Symposium on the Frontiers of Massively Parallel Computation*, pp. 181 – 186, Oct. 1988.
- [128] H. Li and M. Maresca, "Polymorphic-Torus Network," *IEEE Transactions on Computers*, pp. 1345 – 1351, Sep. 1989.
- [129] A. T. Ogielski and W. Aiello, "Sparse Matrix Computations on Parallel Processor Arrays," *SIAM Journal on Scientific Computing*, pp. 519 – 530, May. 1993.
- [130] M. Misra, D. Nassimi, and V. K. Prasanna, "Efficient VLSI implementation of iterative solutions to sparse linear systems," *Prentice-Hall, Inc.*, pp. 52 – 61, ISBN:0-13-473422-X 1990.
- [131] L. H. Ziantz, C. C. Ozturan, and B. K. Szymanski, "Run-Time Optimization of Sparse Matrix-Vector Multiplication on SIMD Machines," *Proceedings of the 6th International PARLE Conference on Parallel Architectures and Languages Europe*, pp. 313 – 322, Jul. 1994.
- [132] B. C. Lee, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "Performance Models for Evaluation and Automatic Tuning of Symmetric Sparse

- Matrix-Vector Multiply,” *Proceedings of the International Conference on Parallel Processing, ICPP’04*, pp. 169 – 176, Aug. 2004.
- [133] M. A. Erle, M. J. Schulte, and J. M. Linebarger, “Potential Speedup Using Floating-Point Decimal Hardware,” *Proceedings of the Thirty Six Asilomar Conference on Signals, Systems and Computers*, pp. 1073 – 1077, Nov. 2002.
- [134] R. D. Kenney and M. Schulte, “High-Speed Multioperand Decimal Adders,” *IEEE Transactions on Computers*, pp. 953 – 963, Aug. 2005.
- [135] I. XILINX, “The XILINX Software Manuals, XILINX 8.1i,” <http://www.xilinx.com/support/library.htm>, 2007.
- [136] Xilinx, “FSL_V20 Xilinx IP Core,” *DS 449 (Product Specification)*, pp. 1 – 8, Dec. 2005.
- [137] J. P. Heron, R. Woods, S. Sezer, and R. H. Turner, “Development of a Run-Time Reconfiguration System with Low Reconfiguration Overhead,” *The Journal of VLSI Signal Processing*, pp. 97–113, May 2001.
- [138] S. M. Scalera and J. R. Vazquez, “The Design and Implementation of a Context Switching FPGA,” *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 15 – 17, Apr. 1998.
- [139] R. Siripokarpirom, “Platform Development for Run-Time Reconfigurable Co-Emulation,” *Proceedings of The Seventeenth IEEE International Workshop on Rapid System Prototyping*, pp. 179 – 185, Jun. 2006.
- [140] I. XILINX, “PlanAhead Tutorial,” http://www.xilinx.com/ise/planahead/PlanAhead_Tutorial.pdf, pp. 1 – 181, Jul. 2007.
- [141] C. M. Huizer, K. Baker, R. Mehtani, J. D. Block, H. Dijkstra, P. J. Hynes, J. A. M. Lammerts, M. M. Lecoutere, A. Popp, A. H. M. van-Roermund, P. Sheridan, R. J. Sluyter, and F. P. J. M. Welten, “A Programmable 1400 MOPS Video Signal Processor,” *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 24.3/1 – 24.3/4, May. 1989.
- [142] P. M. Athanas and H. F. Silverman, “Processor Reconfiguration Through Instruction-Set Metamorphosis,” *IEEE Computer*, pp. 11 – 18, Mar. 1993.

- [143] M. Taylor, "The Raw Prototype Design Document V5.02," *Digest of Technical Papers Proceedings of the IEEE International Conference on Solid-State Circuits*, pp. 1 – 107, Dic. 2005.
- [144] B. Vermeulen, K. Goossens, R. van Steeden, and M. Bennebroek, "Communication-Centric SOC Debug using Transactions," *Proceedings of the European Test Symposium (ETS)*, pp. 61 – 70, May 2007.
- [145] X. Jiang, W. Wolf, J. Henkel, and S. Chakradhar, "A Methodology for Design, Modeling, and Analysis of Networks-on-Chip," *IEEE International Symposium on Circuits and Systems, ISCAS 2005*, pp. 1778 – 1781, May 2005.
- [146] R. B. Kujoth, C. W. Wang, J. J. Cook, D. B. Gottlieb, and N. P. Carter, "A Wire Delay-Tolerant Reconfigurable Unit for a Clustered Programmable-Reconfigurable Processor," *Microprocessors & Microsystems*, pp. 146 – 159, Mar. 2007.
- [147] T. N. Vijaykumar and A. Chishti, "Wire Delay is not a Problem for SMT (in the near future)," *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pp. 40 – 51, Jun. 2004.
- [148] M. J. Wirthlin and B. L. Hutchings, "A Dynamic Instruction Set Computer," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 99 – 107, Apr. 1995.
- [149] M. Berekovic, A. Kanstein, and D. Desmet, "Mapping of Video Compression Algorithms on the ADRES Coarse-Grain Reconfigurable Array," *Proceedings of the Workshop on Multimedia and Stream Processors'05*, pp. 1 – 4, Nov. 2005.
- [150] S. Vassiliadis, S. Wong, and S. D. Cotofana, "The MOLEN $\rho\mu$ -coded Processor," *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 275 – 285, Aug. 2001.
- [151] M. V. Wilkes, "Best Way to Design an Automatic Calculating Machine," *In Report of the Manchester University Computer Inaugural Conference*, p. 16 – 18, Jul. 1951.
- [152] G. Kuzmanov and S. Vassiliadis, "Arbitrating Instructions in a $\rho\mu$ -coded CCM," *In Proceedings of the 13th International Conference FPL'03*, pp. 81 – 90, Sep. 2003.

List of Publications

International Journals

1. H. Calderón C. Galuzzi, G. N. Gaydadjiev and S. Vassiliadis. **High-Bandwidth Address Generation Unit.** *Journal of VLSI Signal Processing*, accepted for publication, to appear in 2008.

Conference Proceedings

1. H. Calderón and S. Vassiliadis. **Reconfigurable Universal SAD-Multiplier Array.** In *Proceedings of the 2nd conference on Computing Frontiers, SESSION: Track 4: reconfigurable computing (part 1)*, ISBN:1-59593-019-1, pages 72–76, May 2005.
2. H. Calderón and S. Vassiliadis. **Reconfigurable Multiple Operation Array.** In *Proceedings of the Fifth International Workshop on Computer Systems: Architectures, Modeling, and Simulation (SAMOS 2005)*, LNCS 3553, pages 22–31, July 2005.
3. H. Calderón and S. Vassiliadis. **Reconfigurable Fixed Point Dense and Sparse Matrix-Vector Multiply/Add Unit.** In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'06)*, pages 311–316, September 2006.
4. H. Calderón C. Galuzzi, G. N. Gaydadjiev and S. Vassiliadis. **High-Bandwidth Address Generation Unit.** In *Proceedings of the 7th International Workshop on Computer Systems: Architectures, Modeling, and Simulation (SAMOS 2007)*, LNCS 4599, pages 263–274, July 2007.
5. H. Calderón, G. N. Gaydadjiev and S. Vassiliadis. **Reconfigurable Universal Adder.** In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*

(ASAP'07), pages 186–191, July 2007.

Technical Report

6. H. Calderón, G. N. Gaydadjiev and S. Vassiliadis. **FPGA based implementation of Reconfigurable Universal Adders.** *Technical Report : CE-TR-2007-01*, pages 1–21, January 2007.

Conference Proceedings (National)

1. H. Calderón and S. Vassiliadis. **Soft Core Processors and Embedded Processing: a survey and analysis.** In *Proceedings of the ProRISC 2005*, pages 483–488, November 2005.
2. H. Calderón and S. Vassiliadis. **The Universal Multiplier.** In *Proceedings of the ProRISC 2004*, pages 341–346, November 2003.
3. H. Calderón and S. Vassiliadis. **Computer Graphics and the MOLEN Paradigm: a survey.** In *Proceedings of the ProRISC 2003*, pages 23–36, November 2003.
4. H. Calderón. **Fuzzy Logic Control for the Kunka Aña Mobile Robot.** In *Proceedings of the ProRISC 2003*, pages 37–42, November 2003.

Samenvatting

In deze dissertatie behandelen we het ontwerp van multi-functionele rekenkundige units, gebruik makend van de meest gebruikte fixed-point cijferrepresentatie. Deze zijn : unsigned, sing-magnitude, fractionele, tien en twee complement notaties. Onze voornaamste doelstelling is om meerdere complexe rekenkundige operaties te kunnen samenbrengen in één enkele, universele rekenkundige unit waarbij getracht wordt zoveel mogelijk hardware te hergebruiken. Meer specifiek proberen we een rekenkundige eenheid te maken voor de 'som van absolute verschillen' (SAD) en vermenigvuldigings operaties (AUSM). Deze eenheid omvat meerdere operaties gebaseerd op multi-operand optellingen, zoals SAD, universele notatie vermenigvuldiging, multiply-accumulate (MAC) en fractionele vermenigvuldiging. Ons AUSM ontwerp hergebruikt op aantoonbare wijze tot 75 % van de aanwezige hardware waarbij de prestaties vergelijkbaar zijn met de snelste, stand-alone ontwerpen die de individuele operaties ondersteunen. Een andere complexe, rekenkundige operatie die wordt bekeken is de matrix vermenigvuldiging. Hierbij hebben we fixed-point matrix vermenigvuldiging eenheden voor dense en sparse matrices in één eenheid ondergebracht. De implementatie op de Xilinx Virtex II Pro geeft prestaties weer tot 21GOPS op een xc2vp100-6 FPGA. In deze thesis stellen we ook een rekenkundige eenheid voor universele optelling voor die zowel optelling als aftrekking van getallen in binaire en Bincary Coded Decimal (BCD) voorstelling mogelijk maakt. Het hardware hergebruik voor deze eenheid bedroeg 40 % en de prestaties lagen hoger dan 82MOPS. Alle beschouwde eenheden vereisen massaal parallelle geheugens die voldoende data throughput moeten garanderen. Daartoe stellen we eveneens een hoog performante adres generator (AGEN) voor die gestoeld is op lage orde, interleaved geheugen. Onze experimenten tonen aan dat de AGEN elke 6 ns een 8 x 32 bit adres kan genereren. Samenvattend, in deze dissertatie hebben we een ontwerp voorgesteld dat toelaat om rekenkundige eenheden met elkaar te laten samenvallen waarbij zowel hoge rekenprestaties als een efficiënt gebruik van hardware voor de beschouwde functies, worden gegarandeerd.

Curriculum Vitae

Daniel Ramiro Humberto CALDERON RO-CABADO was born on the 21th of July, 1964 in La Paz, Bolivia. After his primary and secondary education at “Colegio La Salle”, where he obtained the Baccalaureate degree, he studied at the “Instituto Tecnológico de Costa Rica (ITCR)”, where he graduated as an Electrical Engineer in 1991.



In 1992, he established a consultancy company “BOLTECH” in Cochabamba Bolivia. In the same period he began lecturing at the electronic department of the “Universidad Privada del Valle (UNIVALLE)”. In 1997 he obtained with honors the MSc. in Computer Sciences from the the ITCR (Costa Rica). From August 1997 he started teaching at the “Universidad Privada Boliviana (UPB)” in Cochabamba - Bolivia. In this university, he had been in charge of the relationship of the university with the “Iberoamerican Science and Technology Consortium” (ISTEC) and set up the ISTEC laboratory at the UPB. During 2000-2002, he pursued a second Master degree in Modern Control Systems at the “Universidad Mayor de San Simón” in Cochabamba Bolivia, where the Postgraduate courses were organized by the TU Delft.

In February 2003, he joined the Computer Engineering (CE) lab of Delft University of Technology (TU Delft), The Netherlands, as a researcher, where he had been working towards his Ph.D. degree with scientific advisor prof. dr. Stamatis Vassiliadis. Unfortunately, prof. Vassiliadis passed away in April 2007. His research at TU Delft was supported by the CICAT - TU Delft. This

dissertation contains the outcome of his research activity in the CE Lab, TU Delft, during the period 2003-2007.

Recently, D.R. Humberto Calderón has been employed as a senior engineer and researcher at the “Istituto Italiano di Tecnologia (IIT)” in Genova Italy. Starting from December 2007.

Since 1989 D.R. Humberto Calderón R. is an IEEE member. Also he belongs to the Bolivian Society of Engineering, with National Registry 9.231. His current research interests include: reconfigurable computing, multimedia embedded systems, computer arithmetic, computer architecture, computer organization, intelligent control and robotics.