Università degli Studi di Bologna

FACOLTÀ DI INGEGNERIA

Dottorato di Ricerca in Ingegneria Elettronica,
Informatica e delle Telecomunicazioni

XIX Ciclo

ING-INF/01

Software Tools for Embedded Reconfigurable Processors

Tesi di Dottorato di

Claudio Mucci

Relatore

Chiar. mo Prof. **Roberto Guerrieri**

Coordinatore

Chiar. mo Prof. **Paolo Bassi**

Anno Accademico 2005-2006

Keywords:

Reconfigurable architectures
Programming environment
Application Development
HW/SW Co-Design
Digital Signal Processing

Contents

1 Introduction **1**

2 Reconfigurable computing overview **9**

2.1 Instruction set metamorphosis 9

2.2 Coarse-grained reconfigurable computing 13

2.3 XiRisc reconfigurable processor 20

2.4 DREAM adaptive reconfigurable DSP 25

2.4.1 PiCoGA-III architecture 27

3 Programming tools for reconfigurable processors **31**

3.1 Motivations 31

3.2 Algorithm development on reconfigurable processors (programming issues) 35

3.3 Instruction set extension implementation on a standard compilation tool-chain 38

3.4 Bridging the gap from hardware to software through C-described Data Flow Graphs 42

3.5 Overview of programming tools for reconfigurable processors 46

3.6 Griffy project overview 50

4 Mapping DFG on reconfigurable devices **57**

4.1 ILP exploitation through pipelined DFG and Petri Nets . . . 57

4.2 Instruction scheduling: optimized DFG for pipelined computation 65

4.2.1 Scheduling of direct acyclic graphs 65

4.2.2 Scheduling of data flow graphs 67
4.2.3 Execution-time pipeline management 71
4.2.4 Griffy Front-End architecture 74
4.3 Target-specific customizations and back-end flows 76
4.3.1 DFG mapping for PiCoGA 77
4.3.2 DFG mapping for eFPGA 79

5 Simulation of dynamically reconfigurable processors 85
5.1 Functional simulation 87
5.1.1 Functional emulation 87
5.1.2 Reconfigurable devices management via virtual target 90
5.2 Instruction set extension through dynamic libraries 92
5.2.1 Cycle-accurate simulation model 96
5.2.2 Simulation speed analysis 100

6 Application development on reconfigurable processors 105
6.1 Reconfigurable software development time: hardware and software approaches 111
6.2 Example of application mapping 116
6.2.1 MPEG-2 motion compensation on the XiRisc processor 116
6.2.2 AES/Rijndael implementation on the DREAM adaptive DSP 133
6.2.3 Low-complexity transform for H.264 video encoding 146
6.2.4 H.264 intra prediction with Hadamard transform for 4x4 blocks 162

7 Performance and development time trade-offs 173

8 Conclusions 187

A Griffy-C syntax 191
A.1 Overview 192
A.1.1 Standard Operators 199
A.1.2 Arithmetical Operators 199
A.1.3 Bitwise Logical Operators 200

A.1.4 Direct Assignment 203
A.1.5 Shift Operators 203
A.1.6 Comparison Operators 204
A.1.7 Conditional Assignment 208
A.1.8 Advanced Operators 210
A.1.9 Concatenate operator (#) 210
A.1.10 LUT operator (@) 211
A.1.11 Built-in function as hard-macros 213

List of Figures

1.1 Computational requirements vs. Moore's law and battery storage . . . 2
1.2 Factors considered most important in choosing a microprocessor (source: *J.Turley, "Survey says: software tools more important than chips", Nov. 2005, www.embedded.com*) . . . 5
1.3 Performance vs. Development Time in a commercial DSP (source: *"EFR (Enhanced Full-Rate) vocoder on Dual-MAC ST122 DSP" STMicroelectronics online, www.stm.com*) . . . 7

2.1 PRISC Architecture overview . . . 10
2.2 OneChip architecture . . . 11
2.3 Garp architecture . . . 12
2.4 XiRisc reconfigurable processor architecture . . . 13
2.5 Molen architecture . . . 14
2.6 FPGA integration density (source: R. Hartenstein "Why we need reconfigurable computing education") . . . 15
2.7 MorphoSys architecture . . . 16
2.8 PACT XPP architecture . . . 17
2.9 CHESS architecture and its hexagonal topology . . . 18
2.10 Detailed XiRisc reconfigurable processor architecture . . . 20
2.11 Pipelined Configurable Gate Array (PiCoGA) ver. 1.0 . . . 21
2.12 PiCoGA Reconfigurable Logic Cell (RLC) . . . 22
2.13 Simplified DREAM architecture . . . 25
2.14 Programmable address generator schema . . . 26
2.15 Simplified PiCoGA-III Reconfigurable Logic Cell (RLC) . . . 29

3.1 Performance vs. Time-to-develop design space 34
3.2 Basic software tool-chain extension to support reconfigurability issues 41
3.3 Examples of control and data flow graphs 43
3.4 Griffy Algorithm Development Environment 51
3.5 DFG Description 52
3.6 Example of optimization of routing-only operators 53
3.7 Griffy-C Debugging and Validation Environment 55

4.1 Computation paradigm relaxation preserving the data dependencies 60
4.2 DFG and the corresponding Petri Net representation 62
4.3 Petri Net transition firing 63
4.4 DAG scheduling pseudo-code (with routing only optimization) 66
4.5 Example of ALAP correction for static variables 69
4.6 Simplified DFG scheduling algorithm 70
4.7 Candidates analysis algorithm 72
4.8 Pipeline stage controller simplified architecture 73
4.9 P-block and S-block simplified architecture 73
4.10 Simplified Griffy Front-End architecture 75
4.11 Simplified Griffy flow for PiCoGA-III 76
4.12 PiCoGA-III control unit programmable interconnect 78
4.13 XiSystem SoC architecture 80
4.14 Overall software tool-chain 81
4.15 XiSystem MPEG2 decoder performance 84

5.1 Griffy code viewer 89
5.2 Simplified XiRisc simulation structure 94
5.3 An example of pipeline evolution 97

6.1 Case study: saturating MAC for low bit-rate audio compression 106
6.2 Case study: Griffy-C code for saturating arithmetic 107

6.3 Case study: software pipelining across processor and PiCoGA 108
6.4 Variation of %speed-up wrt T_i and S_i 113
6.5 Variation of speed-up wrt #optimized kernel and local speed-up S_i 114
6.6 Motion estimation 116
6.7 Search path 119
6.8 Absolute Difference (AD) DFG 120
6.9 Concurrent 4-pixel Sum of Absolute Differences 120
6.10 Memory layout 121
6.11 Enhanced search path 123
6.12 Concurrent 4-blocks SAD 124
6.13 Unfolded SAD function based on `sad4blk` 126
6.14 `sad4blk` DFG 127
6.15 `sad4blk` Place & Route 128
6.16 Full-Search workload vs. search window side 131
6.17 Common AES-Round block diagram 137
6.18 Inverse multiplicative on composite fields schemes 139
6.19 AES/Rijndael selected kernel and implementation 140
6.20 Speed-ups *wrt* RISC processor 142
6.21 Throughput vs. interleaving factor 144
6.22 Fast implementation of the 1-D H.264 transform 151
6.23 Fully-unfolded bi-dimensional transform diagram 152
6.24 Partially unfolded 4x4 DCT schema 153
6.25 sub4x4dct rows occupation 154
6.26 Modified sub4x4dct for area optimization 155
6.27 Fully-unfolded inverse 4x4-IDCT basic diagram 156
6.28 Partially-unfolded inverse 4x4-IDCT basic diagram 157
6.29 Modified clipping function structure 157
6.30 Speed-up figure with respect to a RISC processor working at the same frequency 160
6.31 Throughput achieved with respect to interleaving factor . . 160
6.32 Energy efficiency with respect to interleaving factor 161
6.33 Intra prediction modes for 4x4 luma block 162

6.34 PiCoGA SAD structure . . . 164
6.35 DCT and Hadamard transform . . . 165
6.36 1-D Hadamard transform butterfly schema . . . 166
6.37 Fully unfolded 4x4 SATD data flow graph . . . 167
6.38 Partially folded 4x4 SATD block diagram . . . 167
6.39 Shifter register structure used for the matrix transposition . 168
6.40 Optimized SATD mapping . . . 169
6.41 4x4 SAD and SATD speed-up figures with respect to the interleaving factor . . . 170
6.42 4x4 SAD and SATD throughput with respect to the interleaving factor . . . 170
6.43 4x4 SAD and SATD energy efficiency with respect to the interleaving factor . . . 171

7.1 Application development trade-off . . . 177
7.2 Development Time vs Speed-Up percentage . . . 177
7.3 Distribution of speed-up with respect to development time . 178
7.4 XiRisc vs DSP Development Time/Speed-Up analysis . . . 181
7.5 DREAM speed-up . . . 182
7.6 DREAM throughput . . . 183
7.7 DREAM energy efficiency . . . 184

A.1 Multiple entry-point Griffy flow . . . 193
A.2 Concatenate operator . . . 210
A.3 Multiplier chunk . . . 214

List of Tables

4.1 PiCoGA vs. eFPGA computational efficiency comparison . . 82
4.2 Area occupation and working frequency of circuits mapped on the eFPGA 83

5.1 Simulation results (without PiCoGA) 101
5.2 Simulation results (with PiCoGA) 102

6.1 MPEG-2 computation-aware analysis 117
6.2 Test-sequence features 130
6.3 Performances 131
6.4 MPEG-2: final results 132
6.5 AES/Rijndael encoder performance 141
6.6 AES-128 encryption comparisons 145
6.7 sub4x4dct 154
6.8 sub4x4dct 155
6.9 F4x4idct and add4x4 158
6.10 4x4 Sum of Absolute Differences (SAD) 163
6.11 4x4 SATD static performance 168

7.1 Experimental results on application development 175
7.2 XiRisc vs. TI TMS320C6713 Performance Comparison 179
7.3 XiRisc vs. TI TMS320C6713 Performance Comparison 180

A.1 Griffy operators 194
A.2 Typologies of LUTs supported 211

Chapter 1

Introduction

Flexible computational platforms are one of the most important need of the modern electronic marketplace. The growth of non-recurring engineering costs (NREs) coupled with the need of shorter time-to-market impose to look forward, toward flexible solutions. The added capability to update directly on the field or to provide on-the-fly new functionalities makes appealing devices which can both reduce re-design costs and increase the product lifetime. As an example, flexible platforms allow to change the supported standards for telecommunication devices, as cellphones or wireless router, or to build new products when the standard is not well defined, or in the status of draft, in order to match the optimal time-to-market. Furthermore, market convergence toward devices integrating multiple and heterogeneous applications is one of the most important challenge for the consumer electronic scenario. As an example, each smartphone, today, includes office applications, video capabilities, and can work with different wireless communication standards (GSM, UMTS, WiFi and maybe WiMax).

Processor-based embedded systems are becoming wide spread and the term flexibility was often coupled with the presence of a processor and its software programming environment. But, the huge increase of the portable-device market puts pressure on application designers who need to combine computational power, flexibility and limited energy consumption. Modern embedded applications such as wireless communication

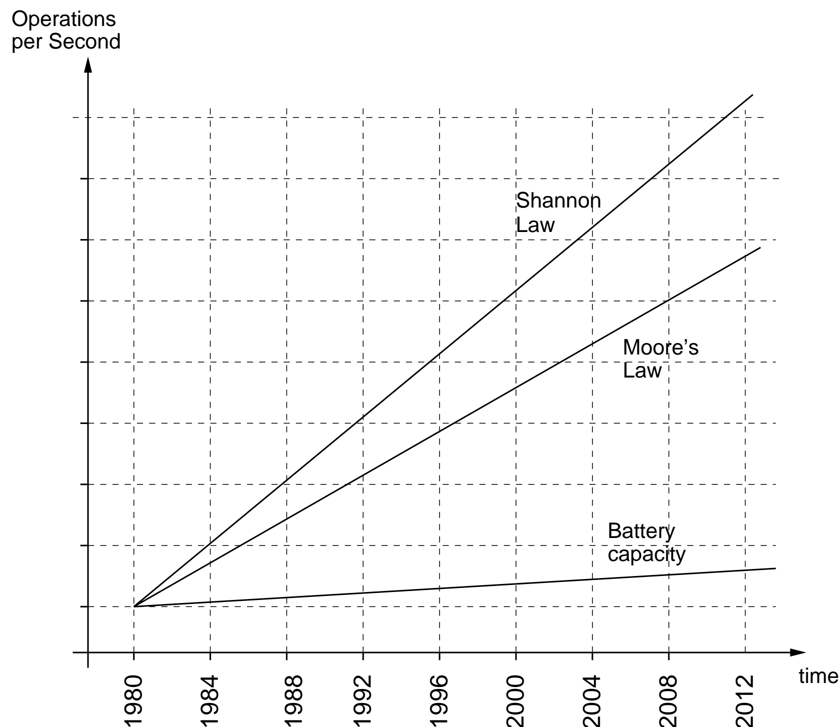


Figure 1.1: Computational requirements vs. Moore's law and battery storage

and portable multimedia require computational power to grow faster than Moore's law and much faster than the energy provided by the batteries for a given application [1], as shown in Fig. 1.1.

In this context and specially for portable low-power applications, designers cannot use the leverage of frequency scaling if they want to meet the performance requirements imposed by quality of service and real-time constraints. The exploitation of instruction level parallelism in many digital signal processors (DSPs) and/or VLIW (Very Long Instruction Word) or superscalar processors for embedded applications is an attempt to tackle the performance gap but usually fails to reduce the energy consumption. Many digital signal processing algorithms require sub-word (e.g. few bits) computations which under-use the common 32-bit datapath of a standard processor [2]. Hence many DSPs provide vectorized processing capabilities, augmenting the instruction set with Single Instruction Multiple Data (SIMD) instructions (like the Intel MMX). On the other hand, microprocessors, whether general-purpose processors or DSPs, remain the most

reusable block in modern systems-on-chips (SoC) and the high-level languages used for programming them are well-known skills among embedded-application developers.

A new processor-based computation paradigm, namely “adaptive computing”, appeared in the early 90s as a promising way to bridge the gap between general purpose microprocessors and application specific integrated circuits (ASICs), in order to support new applications which were both computational intensive and energy hungry. The most appealing idea was to add application specific hardware accelerators in a standard processor architecture (typically a RISC processor) to improve performance on application critical hot-spots, while letting the processor handle the control parts.

It should also be noted that, given a technology node, the area required for a new processor architecture increase by a factor that is greater than the achieved performance improvements. This means that the traditional computing paradigm offered by processors itself loses in computational efficiency (operations per second per mm^2), thus causing an undeniable crisis of standard and well-known devices. State of the art system-on-chips for mobile applications, like ST Nomadik, Philips Nexperia, TI OPAM and Intel PXA, meet performance requirements and power efficiency using the processor (usually an ARM9 core) as a supervisor (for example, the operating system runs on the processor), while the computational intensive parts are commonly demanded to application-specific hardware accelerators. From an engineering point of view, in this way, the effort of accelerating an application focuses on a few computational intensive kernels, thus reducing the time-to-develop.

A wide scenario of adaptive computing approaches has been presented in the literature. We can distinguish among three different approaches:

- *Application-Specific Standard Processors (ASSP)*, which are processors featuring a customized instruction set targeting a given application. Application specific instructions include, for example, the simple multiply-and-accumulate operation in DSPs or the Sum of Absolute

Differences (SAD) used in video encoding motion compensation engines [3].

- *Configurable processors*, which enable SoC designers to rapidly extend a base processor for specific application tasks (e.g. adding custom-tailored execution units, registers, register files and communication mechanisms at the register transfer level (RTL)), thus providing a faster and easier way to build an ASSP [4, 5, 6].

- *(Dynamically) Reconfigurable processors*, which are able to customize the instruction set at execution time by coupling a standard processor core with a run-time programmable device, such as a Field Programmable Gate-Array (FPGA) [30, 10].

While in both ASSPs and configurable processors the instruction set extension is defined at the mask level, thus limiting the device in term of both flexibility and application field, dynamically reconfigurable architectures allow the end-user to meet the requirements of a wide range of applications. Consequently, dynamically reconfigurable architectures are also suitable for use in low volume products as well, since they do not suffer from non-recurring design costs. Furthermore, run-time reconfigurability (also known as on-line reconfigurability) allows one to update the device frequently, thus increasing its lifespan.

In the field of run-time programmable machines, reconfigurable processors form a natural extension to the widely used DSPs or microcontrollers for embedded applications, providing a third trade-off point, in addition to general purpose architectures and dedicated hardware accelerators. However, reconfigurable processors alter the boundary between traditional hardware and software programming, requiring inevitable changes in the programmers' approach and the definition of new design patterns [29]. Algorithm development on reconfigurable processors requires expertise in both hardware and software programming flows and this may prove an obstacle for a community of developers long used to C-based algorithm implementations.

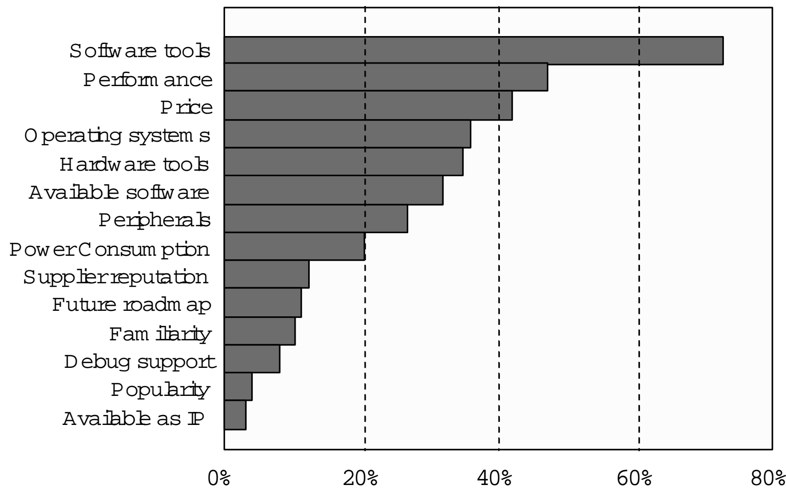


Figure 1.2: Factors considered most important in choosing a microprocessor (source: *J.Turley, "Survey says: software tools more important than chips", Nov. 2005, www.embedded.com*)

According to a survey on embedded development in the telecommunications, automotive, consumer, wireless, defence, industrial, and automation sectors, software tools are considered the most important factor in choosing a microprocessor (see Fig. 1.2). Of course, if we analyze specific sectors, specific parameters such as power consumption for portable devices increase in importance. Nevertheless, for programmers software tools are "the things they touch", the interface with the processor. In the case of reconfigurable processors the importance of software tools grows because of the hybrid nature of these architectures. This is the reason why the lack of user-friendly tools and flows for exploring and implementing the hardware and software portions of an algorithm has caused so much difficulty when developing an application in such architectures [8].

Although not well suited to capture all the parallelism of an application [9], the fact that knowledge of the ANSI-C programming language is widespread among embedded systems and DSPs programmers suggests one should also use it as the application description language for reconfigurable processors. This introduces the problem of translating behavioral

C into some form of HDL description, or directly into hardware (i.e. configuration bits for a run-time programmable device). Unfortunately, these abstraction layers hide many implementation choices from the designer, often making it difficult to obtain high-quality results even with a deep understanding of the tools and the underlying architecture. Despite this, for a wide spectrum of application fields, and thus for a large part of application developers, the availability of a fast and easy way to improve system performance has an impact on the time-to-market, increasing the return on investments. Many reconfigurable processors described in the literature, as well as many start-ups, propose C-based design frameworks in order to cut long-time implementation cycles and/or to reduce the skill gaps for reconfigurable architecture development.

In this thesis will be described a C-based algorithm development environment for reconfigurable processors. It has been successfully applied to the XiRisc reconfigurable processor, coupling a RISC core to a custom-designed mid-grain reconfigurable datapath. It has been also realized a prototype providing the HDL code required to a RISC processor enhanced with a standard embedded FPGA. A C-based configuration flow enables even unexperienced users to efficiently develop algorithms on the reconfigurable processor. Performance improvements of 2-3× can be obtained in 1-2 days of work, without requiring hardware design expertise or awareness of the underlying architecture. Of course, experienced users can achieve far better results, through manual optimizations, just as DSP programmers may optimize at the assembly level so as to obtain the optimal performance. As an example, Fig. 1.3 shows a case-study addressing the relation between development time and performance in the case of a commercial DSP featuring a specific instruction set extension for audio coding applications. Near-optimal performance can be achieved by focusing the implementation effort (mainly spent working with built-in functions, loop restructuring and assembly-level optimization) on less than 25% of the code lines.

Most existing reconfigurable architectures use automatic or semiautomatic C-to-HDL conversion tools to plug into standard synthesis and

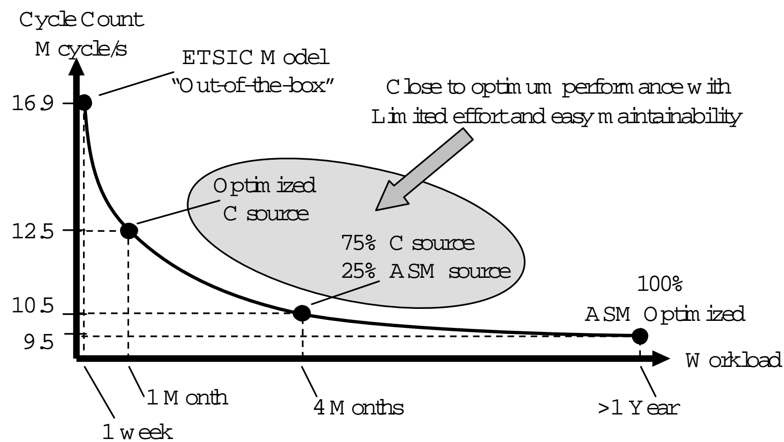


Figure 1.3: Performance vs. Development Time in a commercial DSP (source: *"EFR (Enhanced Full-Rate) vocoder on Dual-MAC ST122 DSP" STMicroelectronics online, www.stm.com*)

Place & Route techniques for configuring the hardware accelerator. The approach proposed in this thesis targets application fields where trading some of the performance speed-up for a higher level of programmability is important. A key contribution of this thesis is that it shows *quantitatively* how much performance one can gain by spending additional time finely optimizing an implementation without ever leaving the purely C-based design environment. It will be shown that knowledge of hardware description languages and hardware design techniques is not required for effective exploitation of a dynamically reconfigurable architecture, especially if the latter has been designed from the beginning to accommodate an efficient software-oriented design flow.

Chapter 2

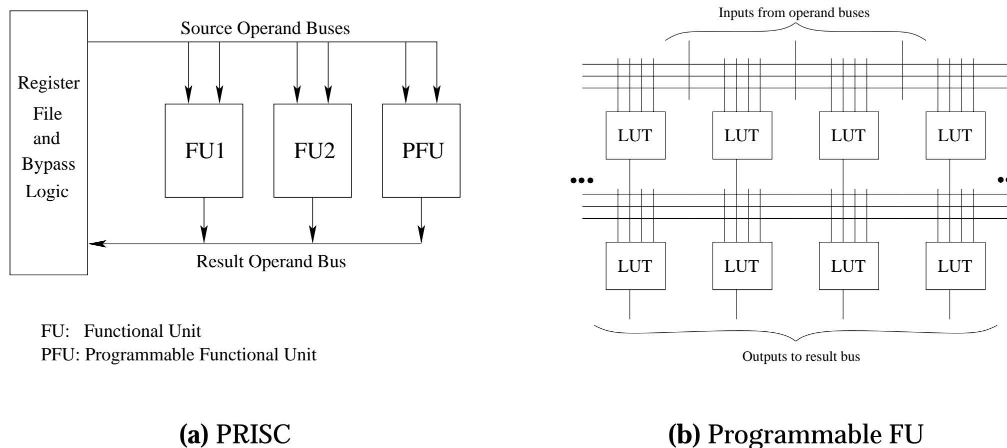
Reconfigurable computing overview

2.1 Instruction set metamorphosis

On 22 May 1999, The Economist (vol. 351, no. 8120, p. 89) reported the following:
"In 1960 Gerald Estrin, a computer scientist at the University of California, Los Angeles, proposed the idea of a fixed plus variable structure computer. It would consist of a standard processor, augmented by an array of reconfigurable hardware, the behavior of which could be controlled by the main processor. The reconfigurable hardware could be set up to perform a specific task, such as image processing or pattern matching, as quickly as a dedicated piece of hardware. Once the task was done, the hardware could be rejigged to do something else. The result ought to be a hybrid computer combining the flexibility of software with the speed of hardware. Although Dr. Estrin built a demonstration machine, his idea failed to catch on. Instead, microprocessors proved to be cheap and powerful enough to do things on their own, without any need for reconfigurable hardware. But recently Dr. Estrin's idea has seen something of a renaissance. The first-ever hybrid microprocessor, combining a conventional processor with reconfigurable circuitry in a single chip, was launched last month. Several firms are now competing to build recon-

figurable chips for use in devices as varied as telephone exchanges, televisions and mobile telephones. And the market for them is expected to grow rapidly. Jordan Selburn, an analyst at Gartner Group (an American information-technology consultancy), believes that annual sales of reconfigurable chips will increase to a value of around $50 billion in 10 years time. *(The Economist: Reconfigurable Systems Undergo Revival)*".

Thanks to the evolution of microelectronics and the enhancement of Field Programmable Gate Array, after 30 years from the Estrin's idea, in 1993, Athenas and Silverman formalized the concept of instruction set metamorphosis or adaptive instruction set proposing the PRISM architecture [16]. Coupling a RISC processor with a Xilinx FPGA the authors realized the first relevant prototype of reconfigurable processor. For the embedded world, the first significant example of processor including run-time programmable hardware in the same chip is probably the PRogrammable Instruction Set Computer (PRISC) [17] proposed by Razdan and Smit one year after.



(a) PRISC **(b)** Programmable FU

Figure 2.1: PRISC Architecture overview

As shown in Fig. 2.1, the PRISC architecture defines a straightforward and efficient way to exchange data with the programmable hardware adopting a schema in which the programmable hardware was embedded in the processor pipeline as the other functional units (Arithmetic Logic Unit, multiplier,...). The Programmable Functional Unit (PFU) has

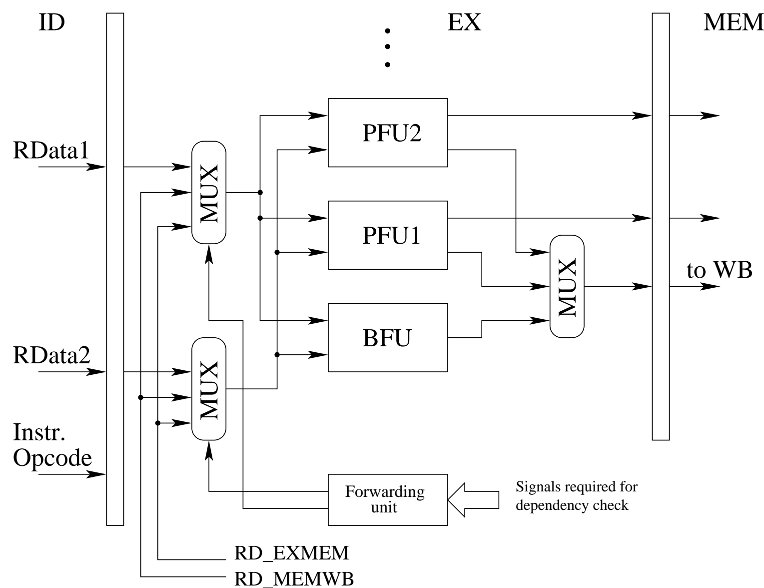


Figure 2.2: OneChip architecture

been designed as a combinatorial matrix of Look-Up Tables (LUTs) interconnected via programmable wires like in FPGA technology. Combinatorial paths limited both the frequency and the size of the PFU. Following an analogue schema, Wittig and Chow proposed the OneChip architecture [18], improving the PRISC proposal with the capability of implementing sequential logic and Finite State Machine (FSM).

One of the most important milestones of reconfigurable computing is the Garp processor [19], developed at the University of California, Berkeley. Garp couples a MIPS processor with a FPGA-like reconfigurable device organized as a datapath (see Fig. 2.3). As for the second release of OneChip, the Garp architecture provides the reconfigurable device the direct access to the memory with an undeniable computational advantage. In fact, while the computation shifts from the processor to the programmable hardware, the access to data long time appeared as a wall (or a bottleneck) for the first generation of reconfigurable processors. In the case of Garp, the reconfigurable array is connected to the processor core as a coprocessor accessed by explicit move operations (move-to, move-from) like that ones required for floating-point units.

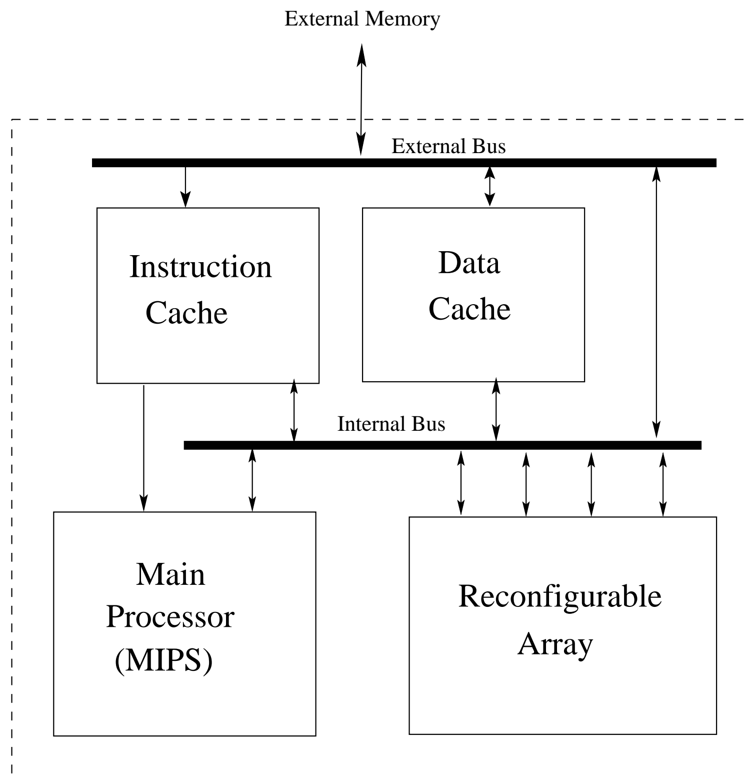


Figure 2.3: Garp architecture

XiRisc reconfigurable processor [66] can be considered the first silicon implementation of custom designed reconfigurable instruction set processor. XiRisc couples a 2-way 32-bit Very Long Instruction Word (VLIW) RISC processor with a custom designed reconfigurable LUT-based datapath (the Pipelined Configurable Gate Array, PiCoGA) integrated in the processor pipeline, as well as the other functional units. The VLIW architecture allows to read up to 4 and write up to 2 registers at once, thus improving the bandwidth between the processor core and the reconfigurable device, although a direct memory access is not provided. As in Garp, the datapath control is performed by a dedicated programmable pipeline manager, that enables the activation of each array row. Fig. 2.4 shows the overall architecture, while section 2.3 provides a detailed description of this architecture and its embedded reconfigurable device Pi-

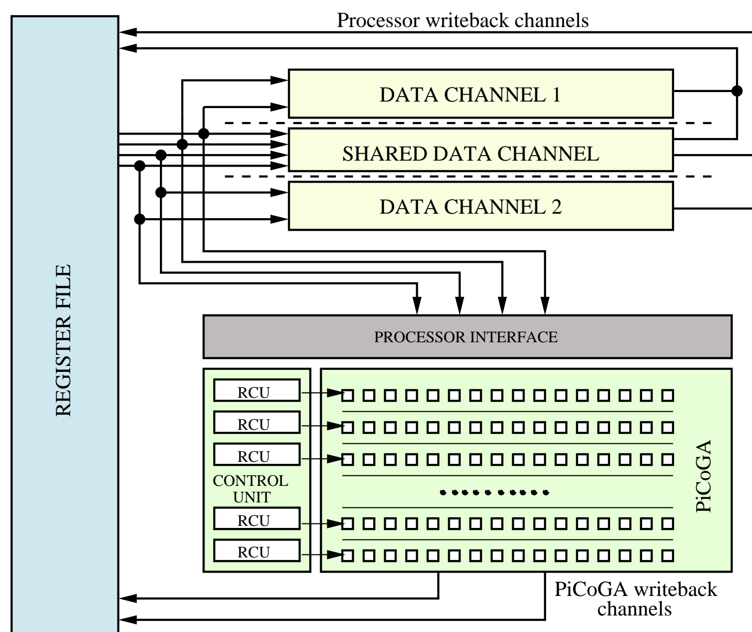


Figure 2.4: XiRisc reconfigurable processor architecture

CoGA.

The Molen [12] polymorphic processor focuses on the architectural formalization of the reconfigurable computation paradigm, with a special glance at programming aspects. The Molen has been implemented on a Xilinx Virtex-II Pro FPGA, utilizing the embedded PowerPC 405 core to allocate, deallocate and execute instructions on the reconfigurable hardware, as depicted in Fig. 2.5.

2.2 Coarse-grained reconfigurable computing

Standard FPGA technology has been the heart and soul of reconfigurable processing pioneers, focused on the formalization of the new computation paradigm. Unfortunately, state of the art FPGAs early appeared as too big, slow and power hungry if compared to application requirement and ASIC-based solutions. The full-flexibility offered from the bit-level programmability introduced too many overhead due to programmable logics,

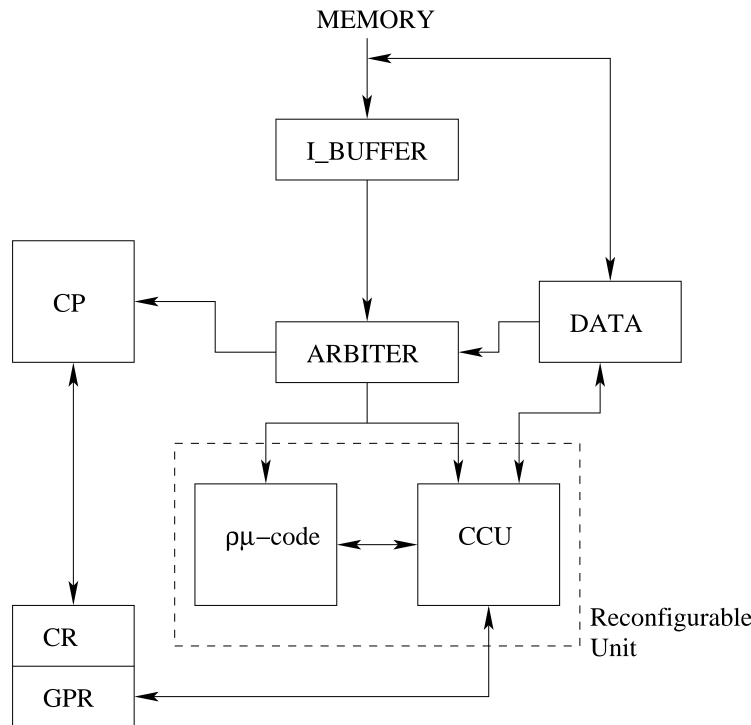


Figure 2.5: Molen architecture

programmable interconnects and Static RAM cells needed to configure all the device. Comparing the number of transistors available for computation to the number of transistors required by a standard FPGA technology we need to remove wiring and reconfigurability overheads resulting about three order of magnitude below the Moore curve. This gap increases since the effective density is reduced by routing congestion that in big devices further decrease the interconnect capabilities, as shown in Fig. 2.6. This is what Hartenstein calls "reconfigurable computing paradox", born to be more efficient than processors in term of operations per second per mm^2, but intrinsically less efficient in term of transistor per mm^2 if compared to the Moore curve.

The need for new programmable devices envisioned in the past years by Nick Tredennick has been accomplished by the proposal of a surprisingly wide scenario of reconfigurable devices trading part of the flexibility

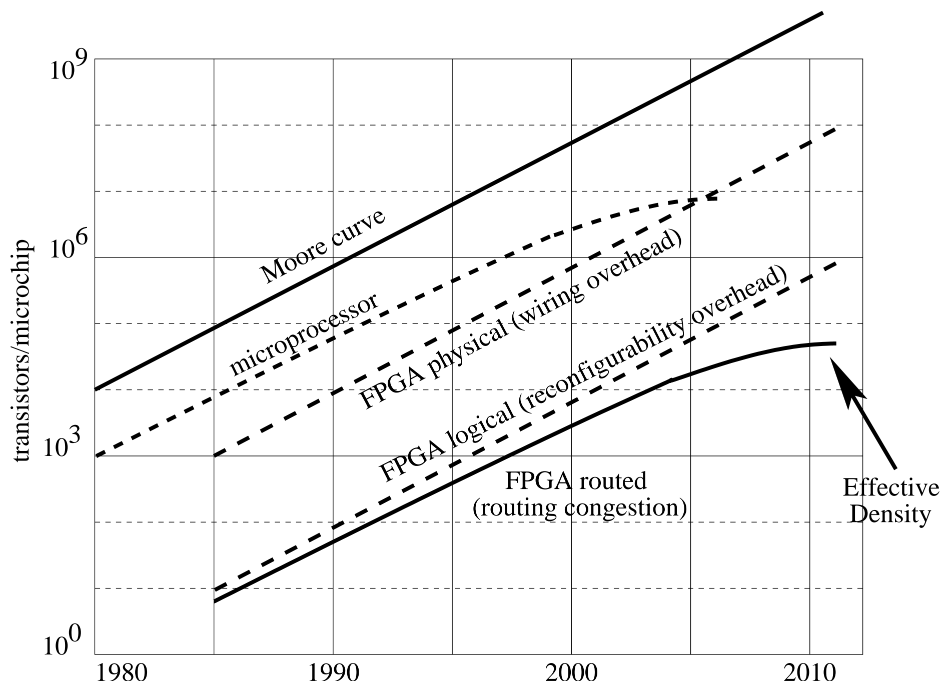


Figure 2.6: FPGA integration density (source: R. Hartenstein "Why we need reconfigurable computing education")

in order to improve hardware efficiency. In its visionary retrospective [30], Hartenstein underlined that "in contrast to FPGA use (fine grain reconfigurable) the area of Reconfigurable Computing mostly stresses the use of coarse grain reconfigurable arrays (RAs) with path-widths greater than 1 bit, because fine-grained architectures are much less efficient because of a huge routing area overhead and poor routability [2]. Since computational datapaths have regular structure, full custom designs of reconfigurable datapath units (rDPUs) can be drastically more area-efficient, than by assembling the FPGA way from single-bit CLBs. Coarse-grained architectures provide operator level CLBs, word level datapaths, and powerful and very area-efficient datapath routing switches."

Many mid and coarse grain devices (but all termed as coarse grain in the Hartenstein's taxonomy) have been proposed from both academia and industry in order to increase the ratio between the grain of the basic logic cell and the programmable interconnects in which the computational logic

is embedded. The computational capability of the basic logic cell shifts from the few LUTs to complete 32-bitwise arithmetic logic units (ALUs). Furthermore, in many cases, interconnect flexibility has been reduced, for example supporting only the connection of nearest rows or among nearest-neighbor cells, in this way also reducing the associated overhead.

PipeRench [46] is one of the firsts and most important reconfigurable devices featuring a datapath structure based on *stripes*. Each *stripe* is composed by arithmetic logic unit, LUTs and a dedicated circuitry to speed-up carry chains. PipeRench introduces the concept of virtual hardware computation by means of fast partial dynamic reconfiguration. The configuration of each stripe can be rapidly changed from the pipeline manager thus allowing to fold deep pipelines on the device.

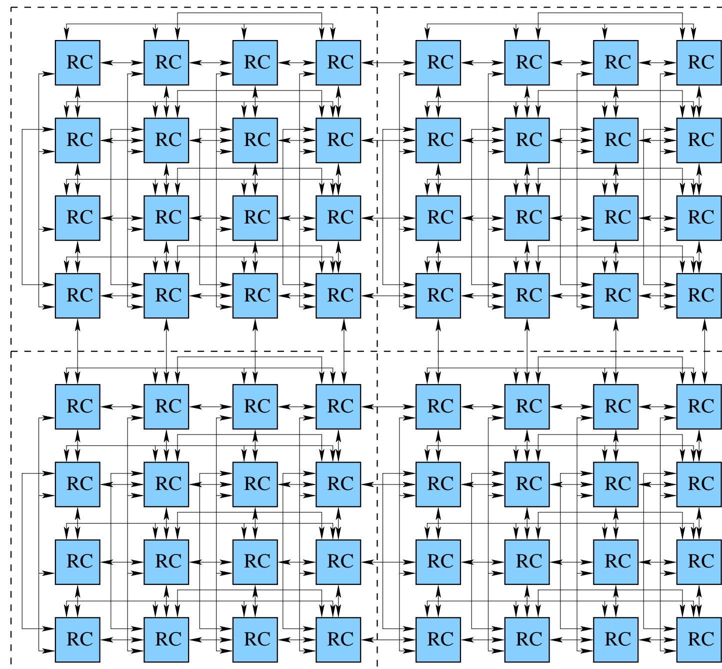


Figure 2.7: MorphoSys architecture

MorphoSys [55] couples a 32-bit RISC core with an 8x8 mesh of 16-bit ALUs with a peculiar interconnect architecture based on nearest-neighbor wires and few regional connections (see Fig. 2.7). In order to reduce the

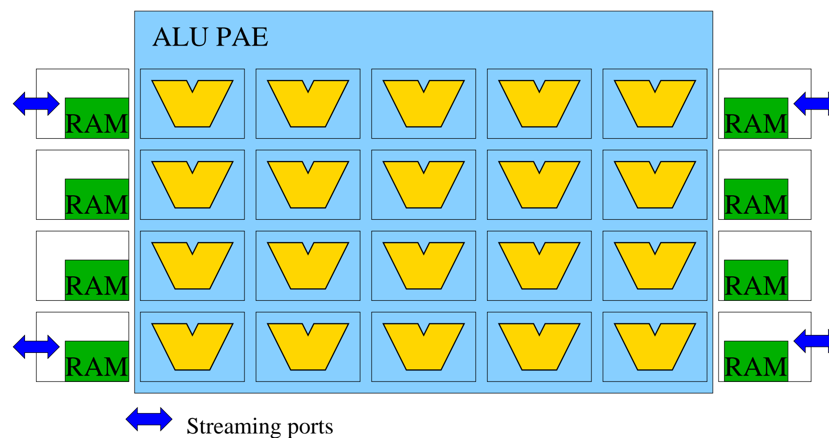


Figure 2.8: PACT XPP architecture

configuration bits, the mesh can be programmed by rows or columns. In other words, each row or column can implement a single instruction multiple data (SIMD) computation. The architecture features a multi-context configuration memory in order to minimize reconfiguration penalty.

The PACT XPP digital signal processor [53] is composed by a matrix of 16-bit Processing Array Elements (PAEs) working as an event-driven data-stream datapath. Internal signals synchronize the data-flow, while the communication is performed by means of packets transmission. Concerning the routing architecture, the array is organized in rows, and the data transfer among successive rows is performed in a synchronous way through dedicated registers. Recently, PACT has introduced also small processor cores based on a simplified 16-bit VLIW structure, in order to achieve better performance figures on control intensive tasks. Figure 2.8 shows the overall architecture.

The CHESS reconfigurable arithmetic array [27], developed from HP Labs and evolved in the *Elixent Ltd. D-Fabrix* [26], is a bi-dimensional mesh of 4-bit ALUs. The principal goals for CHESS were to increase both arithmetic computational density and the bandwidth and capacity of internal memories significantly beyond the capabilities of current FPGAs, whilst enhancing flexibility. For that, a chess-board layout alternating switch-

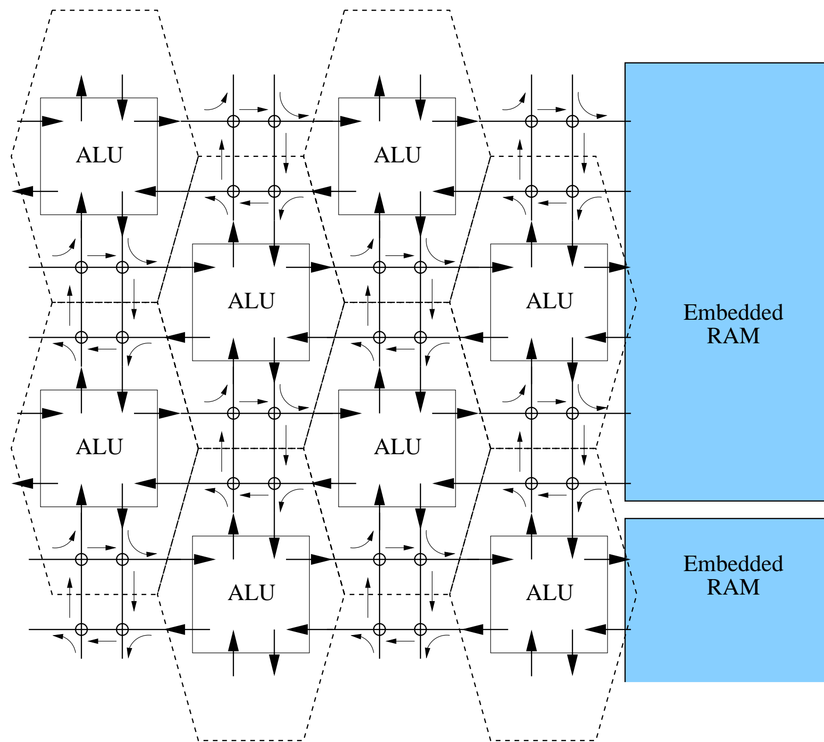


Figure 2.9: CHESS architecture and its hexagonal topology

boxes and ALUs is used as shown in Fig. 2.9. This allows CHESS to support strong local connectivity and communication among ALUs and gives an effective routing network which uses only 50% of the array area, much less than in traditional FPGA structures.

DREAM adaptive DSP [75] is one of the most recent reconfigurable processor coupling a standard RISC core with a pipelined reconfigurable datapath (the third generation of PiCoGA). The reconfigurable device is an important evolution (*if not a revolution*) of the original PiCoGA, augmented with 4-bit ALUs (comprising extended operations like a Galois Fields Multiplier over GF(2^4)) as basic computational blocks in addition to the 64-bit LUTs. This allows DREAM to increase the computational density of the device. Furthermore, the adopted co-processor schema allows the directed access to the local memory sub-system. In particular, a high bandwidth buffer infrastructure has been implemented in order to allow

up to 12 32-bit inputs and 4 32-bit outputs per cycle (the maximum bandwidth of the new PiCoGA device). Section 2.4 shows the detail of this architecture.

The coarsening process of reconfigurable architectures has underlined some interesting proposals in which the basic cell is represented by a small processor. The two main examples are probably the RAW machine [49] from MIT and the PicoChip [25]. RAW, acronym of Reconfigurable Architecture Workstation, provides a RISC multiprocessor architecture composed of nearest neighbor connected 32-bit modified MIPS R2000 microprocessor tiles. Each processor features 6-stage pipeline with ALU, floating point and 32 Kbyte SRAM. PicoChip is a massively parallel array of 430 heterogeneous processors linked by a deterministic high-speed switching matrix. About 230 processors include multiply and accumulate functionalities, but the characteristics and instruction set of elements should include support for specialist operations such as *spread*, *de-spread* or *compare-add-select*.

Specialization of computational blocks is thus an undeniable trend of reconfigurable computing, similarly at the specialization that DSPs provided with dedicated instruction set extension. The main goal has been and will be to reduce the impact of reconfiguration in term of area. Instead of application specific devices, we can term this approach as field-specific since the flexibility offered by reconfigurable approaches appears higher than that one offered by processor based systems augmented with dedicated circuits. In any case, heterogeneity, additional interconnect constraints, as well as special and complex functionalities have an undeniable impact on the programmability of the device and the efficiency in which the devices can be used from programmers, as will be discussed in the next chapter.

2.3 XiRisc reconfigurable processor

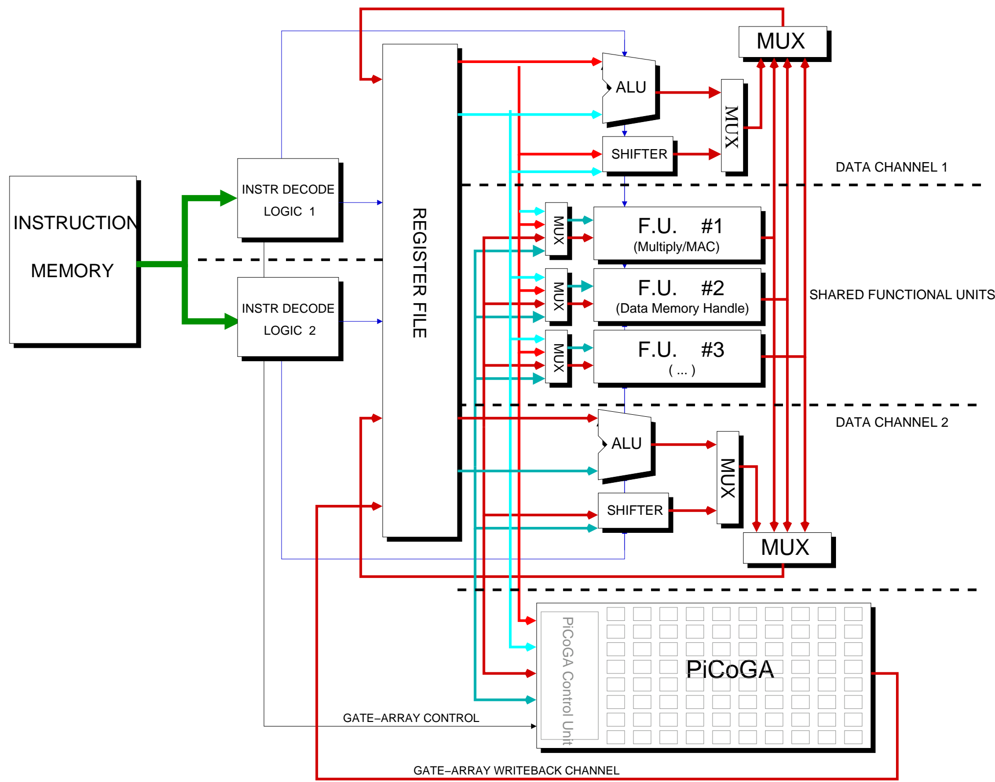


Figure 2.10: Detailed XiRisc reconfigurable processor architecture

The XiRisc reconfigurable processor [66, 68] (Figure 2.10) is a 2-issue Very Long Instruction Word (VLIW) RISC architecture, with two 32-bit data paths, featuring a fine grain reconfigurable functional unit (a PiCoGA, *Pipelined Configurable Gate Array*) that allows the user to dynamically adapt the instruction set to the application workload. PiCoGA is a multi-context array of 24 rows, each of them composed of 16 fine grain Reconfigurable Logic Cells (RLCs), including four-input 16-bit look-up tables and dedicated logic to support the efficient implementation of both arithmetic and logic operators. Programmable interconnects allow point-to-point bit-level communication using the island-style topology showed in Fig. 2.11. In order to reduce the area overhead due to programmable interconnects, the

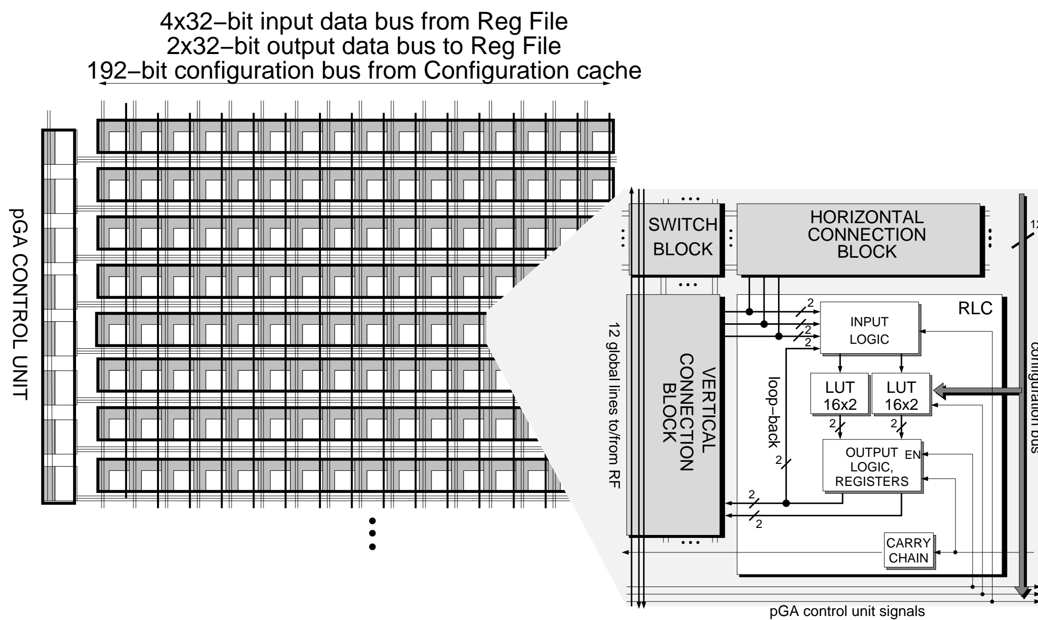


Figure 2.11: Pipelined Configurable Gate Array (PiCoGA) ver. 1.0

routing topology features 2-bit granularity, which is relaxed to 1-bit only at level of the connect blocks. Fig. 2.12 shows the detailed RLC architecture.

XiRisc architecture exploits an assembly-level mechanism to add customized instructions, called PiCoGA operation or `pgaop`, which can replace on average 10-40 assembly instructions, and up to ~400 when a deep hardware approach (involving for example a synthesis step) is adopted. The PiCoGA implements pipelined instructions using a dataflow paradigm [63]. As will be described in the next chapters, customized instructions are extracted, based on user annotations, from ANSI-C code. The Griffy compiler translates them into data-flow graphs (DFGs) which are thereafter mapped on the PiCoGA. From the programmer's point of view, `pgaops` are application-specific *intrinsics* (as pseudo-function calls) in C code or assembly instructions which trigger PiCoGA computations. In this way, the user can still utilize a C-based globally imperative description style for the implemented code.

The PiCoGA reconfigurable device is integrated as a Functional Unit

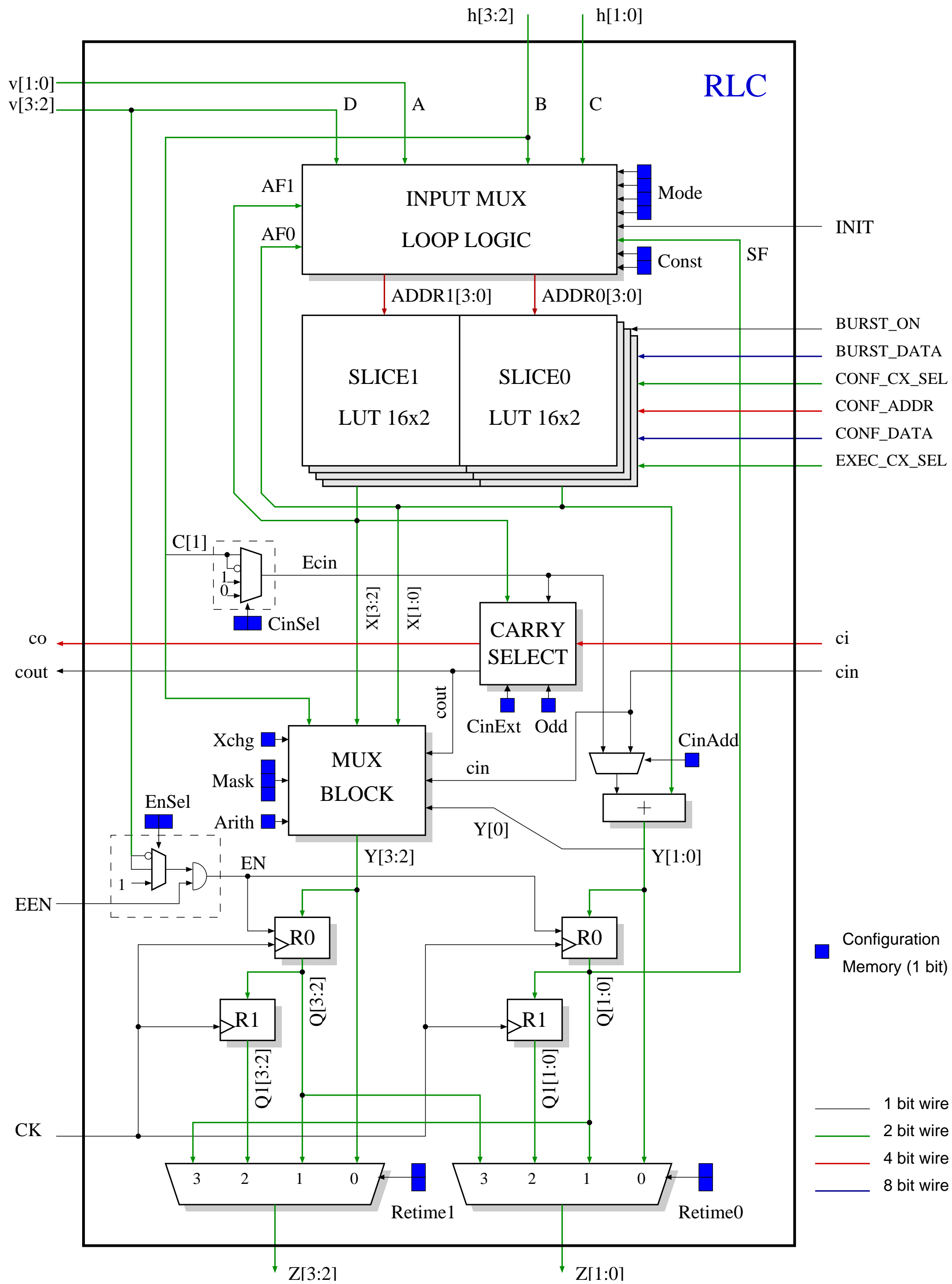


Figure 2.12: PiCoGA Reconfigurable Logic Cell (RLC)

(FU) of the processor core, thus reducing communication overheads to and from other FUs. On the other hand, a register file based communication could be a bottleneck for applications in which high degree of data parallelism can be exploited by streaming or by vectorized direct memory access. The PiCoGA can load up to 4 `pgaops` for each of its 4 configuration contexts, and operations loaded in the same context can be executed concurrently. Embedded hardwired control logic handles conflicts on write-back channels when various `pgaops` need to write on the processor register file. Furthermore, PiCoGA can operate concurrently with the other functional units of the processor, since the data flow consistency is ensured by a register locking mechanism.

Starting from a C source code, the compiling tool builds a pipelined DFG by scheduling instructions. It then maps:

- *DFG-node functionalities* on the PiCoGA RLCs;
- *pipeline management* on a row-based dedicated control unit that enables execution of the mapped pipeline stages [70].

Nodes in the DFG are functional operations mapped onto the device resources (e.g., addition/subtraction or a bitwise logic operation). The pipeline is then built through operations scheduling of the C-level DFG representation. As described in [64] (and explained in the following), a dataflow graph represents dependencies among computational nodes through the data dependencies graph. A pipelined data-flow computation, including both data dependencies and resource constraints, can be modelled using synchronous Petri-Nets [65, 77]. In this model both data dependencies and resource constraints are represented by arcs and tokens and each computation transition fires when all input arcs have a token and a token for each output arc has been produced. A set of transitions which fires simultaneously is also called a *step*.

Following this elaboration pattern, a dedicated programmable control unit is used to handle the pipeline activity, triggering the DFG nodes when all necessary resources are ready and stalling when they are unavailable.

In order to minimize its area occupation, one row control unit (RCU) is dedicated to each array row, so that the minimum granularity for RLC activation is 16. More than one PiCoGA row could be used to build a wider pipeline stage. On the other hand, cascading more than 1 RLC in a single pipeline stage is often impossible because of the fixed high working frequency ($\sim$ 166-200MHz) constraint. When a pipeline stage performs a computation, the control unit exploits a dedicated programmable interconnection channel to send tokens to predecessor and successor nodes [70].

2.4 DREAM adaptive reconfigurable DSP

DREAM architecture [75] is a dynamically reconfigurable platform coupling the PiCoGA-III reconfigurable device with a RISC processor using a loosely-coupled memory mapped co-processor schema. A high bandwidth memory sub-system provides/receives data to/from PiCoGA-III allowing one to both maximize the throughput and interface the DREAM architecture with for example external computational blocks. Figure 2.13 shows the simplified DREAM block diagram.

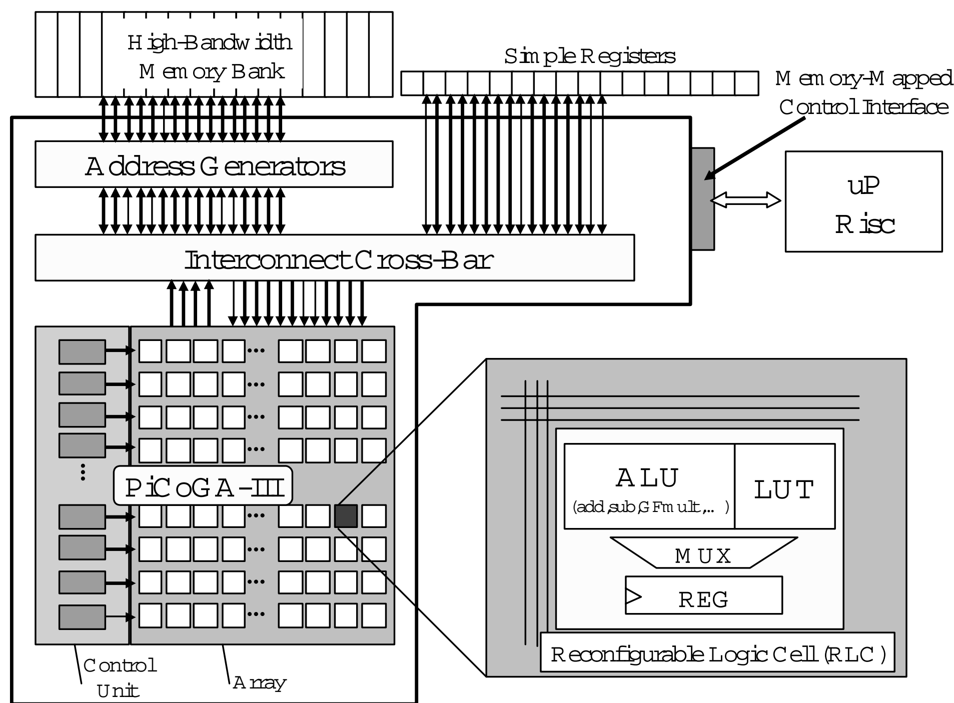


Figure 2.13: Simplified DREAM architecture

The processor, a 32-bit RISC core with 4+4Kbyte of data/instruction memory, is responsible of DREAM management, although it could be also used to implement portions of applications, such as the control part of the code. The high bandwidth memory sub-system is composed of 16 4Kbyte 32-bit memory banks, each of them accessed independently to the other ones by programmable address generators. A fully-populated interconnect cross-bar allows the user to modify the connection with PiCoGA-

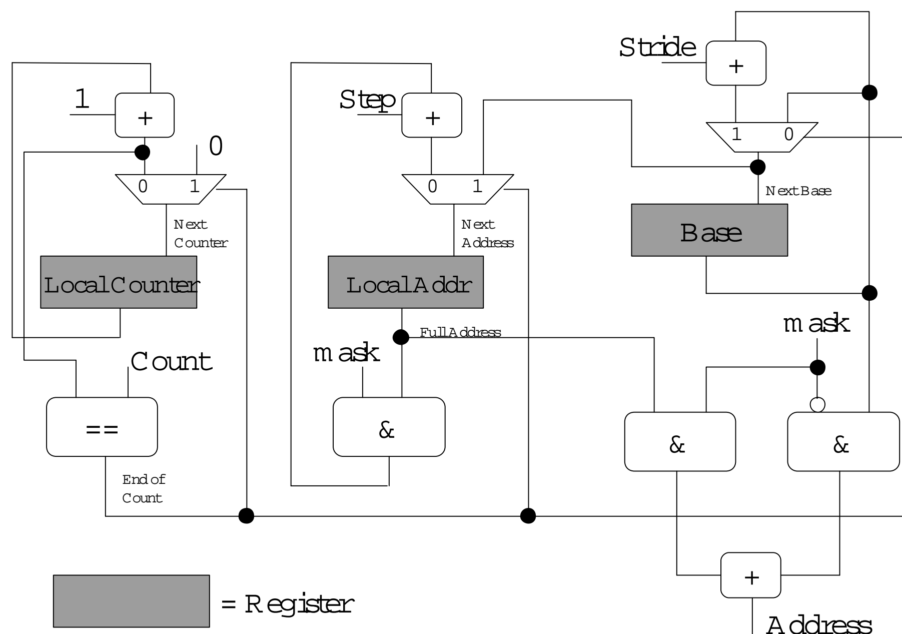


Figure 2.14: Programmable address generator schema

III I/Os (12 32-bit inputs and 4 32-bit outputs). A 64-entry configuration cache is provided for the interconnect, allowing to switch among different connection topologies without any additional overheads, while the same is not provided for the address generators. Furthermore, an additional simple 32-bit register file is provided for local data, synchronized with PiCoGA-III by a register locking mechanism. Concerning the programmable address generators, it has been introduced the capability of handling power-of-2 modulo addressing, in addition to standard step and stride addressing modes. Fig. 2.14 shows the block diagram of the address generators. When *mask* is zero, for each read/write request, the *local address* is initialized to the *base address* and is incremented by *step* (that could be negative) for *count* times. When *count* operations are performed, the *base address* is incremented by *stride* (that could be negative). If *mask* is not set to zero (and features only one transition from 0 to 1, as in 0b00001111), it

allows to perform a selective bit-wise or between the local counter and the base address. Since the local counter updating is *mask*ed, the *mask* allows to wrap around the counting on a power-of-2 sub-buffer.

2.4.1 PiCoGA-III architecture

The PiCoGA-III is a programmable gate array especially designed to implement high-performance algorithms described in C language. The focus of the PiCoGA-III is to exploit the Instruction Level Parallelism (ILP) available in the innermost loops of a wide spectrum of applications, including multimedia, telecommunication and data encryption. From a structural point of view, the PiCoGA-III is composed of 24 rows, each of them implementing a possible stage of a customized pipeline. Each row is composed of 16 Reconfigurable Logic Cells (RLC) and a configurable horizontal interconnect channel. Each RLC includes a 4-bit ALU that allows to efficiently implement 4-bitwise arithmetic/logic operations, and a 64-bit look-up table in order to handle small hash tables and irregular operations hardly describable in C and that traditionally benefit from bit-level synthesis. Each RLC is capable of holding an internal state, as the result of an accumulation, and provides fast carry chain propagation through a PiCoGA row. In order to improve the throughput, the PiCoGA supports the direct implementation of Pipelined Data-Flow Graphs (PDFGs), thus allowing to overlap the execution of successive instances of the same PGAOP (where a PGAOP is a generic operation implemented on the PiCoGA). Flexibility and performance requirements are accomplished handling the pipeline evolution through a dynamic data-dependency check performed by a dedicated Control Unit.

Summarizing, with respect to a traditional embedded FPGAs featuring homogeneous island-style architecture, the PiCoGA-III is composed of three main sub-parts:

- A homogeneous array of 16x24 RLCs with 4-bit granularity (capable of performing operations, for example, between two 4-bitwise variables) and connected through a switch-based 2-bitwise interconnect

matrix;

- A dedicated Control Unit which is responsible to enable the execution of RLCs under a dataflow paradigm;
- A PiCoGA Interface which handles the communication from and to the system (data availability, stalls generation, and so on).

In terms of I/O channels, the PiCoGA-III features 12 32-bit inputs and 4 32 -bit outputs, thus allowing for each PGAOP to read up to 384 bits and to write 128 bits per cycle. The PiCoGA-III is a 4-context reconfigurable functional unit capable of loading up to 4 PGAOPs for each configuration layer. PGAOPs loaded in the same layer can be executed concurrently, but a stall occurs when a context switch is performed. The main features of the PiCoGA architecture are:

- A fine grain configurable matrix of 16x24 RLCs
- A reconfigurable Control Unit, based on 24 Row Control Units (RCUs) that handles the matrix as a datapath.
- 12 primary 32-bit inputs and 4 primary 32-bit outputs
- 4 configuration contexts are provided as a first-level configuration cache
  - only 2 clock cycles are required to change the active context (context switch)
  - only 1 configuration context can be active at a time.
- Up to 4 independent PiCoGA operations can be loaded in each context, featuring partial run-time reconfiguration.

Each RLC can compute algebraic and logic operations on 2 operands of 4 bits each, producing carryout and overflow signals, and a 4-bit result. As a consequence, each row can provide a 64-bit operation or 2 32-bit operations (or four 16-bit, eight 8-bit operations, and so on). The cells communicate through an interconnection architecture with a granularity of 2 bits.

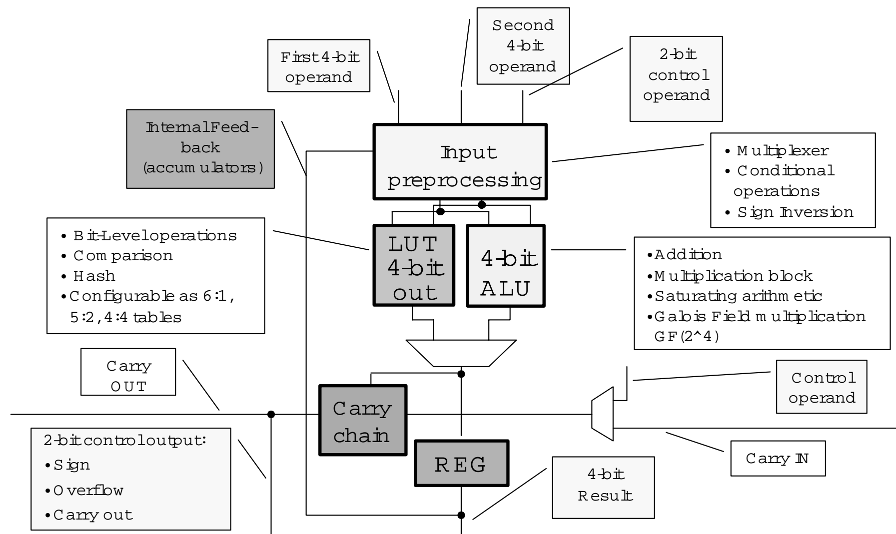


Figure 2.15: Simplified PiCoGA-III Reconfigurable Logic Cell (RLC)

Each task mapped on the PiCoGA is defined PGAOP. The granularity of a PGAOP is typically equivalent to some tens of assembly operations. Each PGAOP is composed by a set of elementary operators (logic or arithmetic operations) that are mapped on the array cell.

Each PiCoGA cell also contains a storage element (FF) that samples the output of each operation. This storage element cannot be bypassed cascading different cells, since the constant frequency of work featured by PiCoGA. Thus PiCoGA can be considered a pipelined structure where each elementary operator composes a pipeline stage. Computation on the array is controlled by a RCU which triggers the elementary operations composing the array. Each elementary operation will occupy at most a clock cycle. A set of concurrent (parallel) operations forms a pipeline stage.

The internal architecture of the Reconfigurable Logic Cell is depicted in Fig. 2.15. Three different structures can be identified:

- The input pre-processing logic, which is responsible to internally route inputs to the ALU or the LUT and to mask them when a con-

stant input is needed

- The elaboration block (ALU & LUT), which performs the real computation based on the operation selected by the RLCop block
- The output manager, which can select outputs from the ALU, the LUT, and eventually from the carry-chain and synchronizes them through Flip-Flops. The output block samples when receives the Execution Enable from the control unit. Therefore the control unit is responsible for the overall data consistency as well as the pipeline evolution.

Operations implemented in the ALU & LUT block are:

- 4-bitwise arithmetic/logical operations eventually propagating a carry to the adjacent RLC (e.g. add, sub)
- 64-bit lookup tables organized as:
  - 1-bit output 4/5/6-bit inputs
  - 2-bit outputs 4/5-bit inputs
  - 4-bit outputs 4-bit inputs
  - a couple of independent lookup tables featuring respectively 1-bit output 4-bit inputs, and 2-bit outputs 4-bit inputs.
- 4-bit multiplier module; more in detail, it is a multiplier module with 10-bit (in case of A * B. 6 bit are for the operand A and 4 bit for the operand B) of inputs and 5-bit output, including 12 Carry Select Adder and specifically designed to efficiently implement small and medium multiplier on PiCoGA resource.
- 4-bit Galois Field Multiplier GF(2^4), with irreducible polynomial $x^4 + x + 1$.

Chapter 3

Programming tools for reconfigurable processors

3.1 Motivations

In the past, the term flexibility has often been linked with the software implementation offered by general-purpose microprocessors, whose computational model aims at:

- modifying the task (the algorithm) by changing a set of instructions (the program) in a read/write memory.
- implementing the algorithm using a small number of general computing resources, roughly corresponding to the assembly instructions, which are reused in the course of time.

This kind of computation is called *temporal computation* and rapidly shows its limitations when algorithm operations fail to match the hardware computational resources [11]. In reconfigurable architectures the application features are exploited through configuration of customizable hardware, thus improving the match between processor capabilities and algorithm requirements. Unlike the temporal computation of traditional processors, reconfigurable processors use a *spatial* model of computation that (ideally) perfectly matches the algorithm, in terms of both available

parallelism and computation granularity. Significant performance speed-up and energy reduction can be achieved on critical kernels compared to standard architectures [7].

Unfortunately, programmable fabrics are less efficient and require far more additional area than dedicated circuits which exploit the *spatial* model of computation as well, but lack flexibility. The silicon cost is often a strong limitation for reconfigurable architectures being proposed to the consumer market. One acceptable trade-off is only to exploit the spatial computation on critical kernels of applications, thus reducing area requirements. Non-critical computations or control-dominated tasks (which often show a very small degree of instruction level parallelism) are efficiently mapped on the standard processor core, taking advantage of its software programmability and shortening the overall development time. According to Amdahl's law and the 90-10 rule (90% of time is spent executing 10% of the lines of a code [28]), performance can be roughly enhanced up to one order of magnitude when implementation efforts focus on the identification and improvement of few critical kernels.

A typical flow for the development of applications on a reconfigurable processor can be based on a common processor-oriented tool-chain augmented in order to handle the instruction set extension. Starting from a high-level description language, the developer needs to partition the application code between hardware and software portions, typically guided by simulation and profiling back-annotations. The partitioning is an iterative process, implying refinements and modifications of a given implementation in order to exploit as much as possible the space-based computation. When the partitioning is decided, the programmer (possibly helped by tools and utilities) describes the hardware part in a proper language, while the interaction between the processor and the reconfigurable hardware can be accomplished by means of built-in functions and/or assembly inlining.

When a portion of code is considered suitable for hardware implementation, it is translated into the description language used as the entry-point for the reconfigurable device. A hardware-specific tool-chain then maps

the description in the device, providing the configuration bit-stream. The translation can be performed by re-writing the code from scratch (for example, the algorithm is completely rewritten in VHDL/Verilog) or can be assisted by tools. Automatic high-level language translation relieves the user from the burden of learning a HDL language, introducing an abstraction layer that however hides many details of implementation, making it hard to handle the performance accurately. On the other hand, when the level of abstraction is tightly linked to the underlying hardware, the time spent on optimizing an application grows, as do the skills required.

Processor-based computation allows the designer to exploit instruction level parallelism (ILP), while a hardware-oriented approach allows one to match application requirements perfectly. If we consider the development time spent in obtaining a specific implementation, processor-based approaches typically require a matter of minutes to compile a given source code, or a few days to optimize critical parts at the assembly level and some weeks to maximize performance using manually-programmed high-end VLIW DSPs. Of course, use of application-specific IPs or manual optimization of an assembly code demands a deep knowledge of the underlying architecture. In the hardware approach, the implementation of a given application in an ASIC takes a long time, it being necessary to describe the algorithm in an optimized RTL HDL taking into account critical paths, for example, after both physical synthesis and place-and-route.

These preliminary considerations are summarized in Figure 3.1, where the performance improvement of a hardware implementation in terms of execution time can be more than 2 orders of magnitude higher than a software one. While a processor-based fully-software flow allows the designer to match the ILP of a given application, a hardware-oriented approach requires one order of magnitude more time-to-develop to achieve the best performance. In the middle of this design space reconfigurable platforms should find their place. The definition of design patterns and frameworks for reconfigurable processors enabling unexperienced users to build applications defines an intermediate "optimization" curve in which the design space moves from software to hardware, in terms of both performance

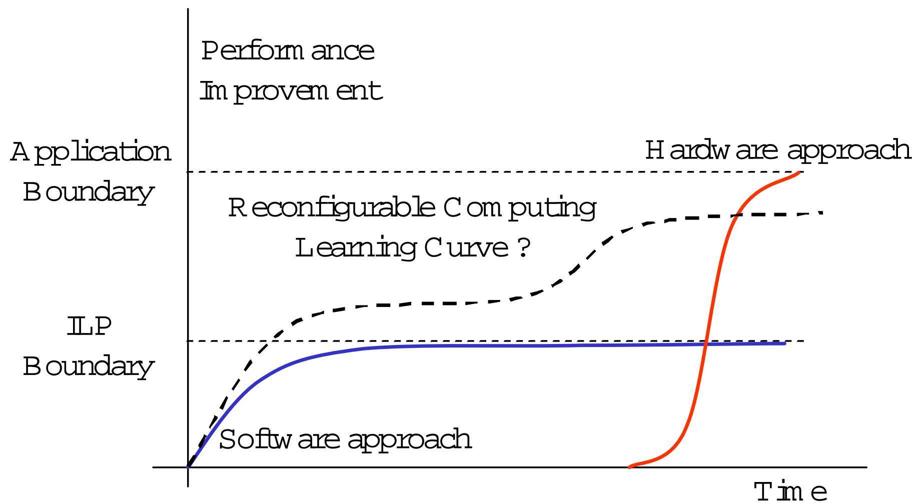


Figure 3.1: Performance vs. Time-to-develop design space

expected and skills required to obtain it.

A further interpretation of these curves can be given in terms of cost modelling. Few works have attempted to examine the impact on costs of hardware/software trade-offs in embedded-system co-design [84, 85, 86, 87]. While the software cost is estimated using standard models, such as COCOMO [82], in these works the hardware cost is mainly based on COTS (Commercial Off-The-Shelf) and libraries of functions. The customization of a reconfigurable processor, despite involving hardware concepts, is not well suited for modelling as a standard hardware development, because it requires an existing component (the reconfigurable processor) to be programmed rather than a new hardware component to be designed. On the other hand development on a reconfigurable processor needs to take into account many more details than a standard processor. For example, in the case of the DSP in Fig. 1.3, better performance can be achieved by spending additional time programming the processor at a lower level of abstraction. Hence, the cost model is characterized by two calibration factors [83] that describe the *average number of code lines per function point*, one for the part of the application written in C and another one for the part written in assembly. The total development time depends on the respective percentage of function points implemented in C and assembly. The same approach

holds for reconfigurable processors, but in this case the specific tools and languages for the given reconfigurable logic need to be considered in addition to C and assembly.

3.2 Algorithm development on reconfigurable processors (programming issues)

Processor-based system-on-chips (SoC) are becoming the most popular way to perform computation in the electronic marketplace. Today, at least one processor is present in every SoC in order to handle in a simple fashion the overall system synchronization, be it provided by the operating system functionalities (i.e. multitasking management, real-time issues) or be it required for I/O communications. Usually, the processor (i.e. ARM9, PowerPC, MIPS, . . .) is not the main responsible of the computation that is demanded to high-performance co-processing engines. Depending on the application constraints and the flexibility required, computation intensive parts are implemented on dedicated hardware accelerators (when non-recurring costs allow that) or on application-specific digital signal processors (DSPs). In this context, high-end DSPs are proposed as a way to match flexibility requirement (since they are software programmable) with high performance. Architectures like the Texas Instruments OMAP or the STMicroelectronics STW51000 (also known as GreenSIDE) are state of the art examples of commercial ARM-based SoCs powered with one or more DSPs *plus* one or more dedicated hardware accelerators. One of the most interesting trend in the field of high-performance SoCs is represented by the introduction of dynamically (or run-time) reconfigurable hardware (i.e. embedded FPGAs, reconfigurable data-path and reconfigurable processors) in substitution of the constellation of DSPs and/or dedicated hardware accelerators today necessary to match constraints in term of performance and energy consumption [1, 2, 7, 8, 11, 10]. In general terms, the exploitation of such kind of architectures implies the capability to tailor the SoC functionalities around the computational requirements of

a given application. This can be seen as an instruction set extension of the main processor (e.g. the ARM in previously cited examples), being the reconfigurable hardware a run-time extension of the baseline computation engine.

As for DSPs and dedicated accelerators, the exploitation of any degree of parallelism at bit-, word-, instruction-, iteration- and task-level is the control lever for the effective utilization of the reconfigurable hardware. This implies for the programmer a deep knowledge of both application and system architecture to well understand how to partition and how to map algorithms over the different available computational resources. On the other hand, this also implies for the programmer the capability to investigate a hybrid design-space including both software and hardware concepts, requirement not so usual for application developer long used to C/assembly design environments. With respect to mask-time programmable hardware accelerators, reconfigurable computing offers the programmer the capability to design its proper extensions in order to satisfy application-specific requirements. Therefore, the capability of providing soft-hardware (or hardware programmable as software) is probably the most important point to enable the large market of application developers to use effectively a reconfigurable device [29].

In the past, the action of targeting a reconfigurable device borrowed tools and methodologies from FPGA-based design (with hand-coded RTL HDL), although it was clear from the beginning the severe lack of user-level programmability coupled to this approach. The utilization of C language has been seen as the most promising way to approach the customers at the reconfigurable proposal. On one hand, C dialects have been presented including entire new object classes dedicated to hardware design, like in System-C or Handel-C. This kind of approach move the C toward the hardware design making the hardware description friendlier through a C-based language that basically becomes another HDL, therefore requiring hardware skills to the developers. On the contrary, a more promising approach is to use standard ANSI C code and translate it into some kind of RTL, thus requiring a sort of C-to-RTL hardware compiler. Com-

panies like Celoxica, Mentor Graphics, Impulse, Altium and CriticalBlue offer stand-alone C-to-RTL and/or C-dialect-to-RTL synthesizers that can be integrated in standard flows for FPGA and that were used in many works on reconfigurable system implemented using commercial FPGAs.

For embedded applications, the reconfigurable device is a part of an usually complex system with a rigid budget in term of area and cost. As underlined in the previous chapter, this precludes the utilization of standard FPGAs, since they are too area demanding for the embedding and not so appealing for the final implementation of the whole system (in term of performance, power consumption and costs). Reconfigurability is then provided through embedded-FPGAs (small FPGAs suitable for the embedding in a SoC), reconfigurable data-paths and reconfigurable processors that offer flexibility under typically hard constraints in term of area. The limitation in terms of area is accomplished by reducing the number of programmable elements and equivalent KGates available, but while a stand-alone high-end FPGA requires some hundreds of mm^2, reconfigurable devices show an area occupation ranging from few mm^2 to some tens. Even if much less than FPGAs, the area occupation of reconfigurable devices is very often considered huge from SoC designers. This means that the reconfigurable device needs to be as small as possible, while the configuration efficiency must grow up to the peak performance offered by the device.

On the architecture side, area limitation can be accomplished by an accurate trade-off between logic and interconnects. For example, in island-style programmable architectures it is possible to achieve better area figures increasing the grain of the basic logic element with respect to the interconnect structure, or decreasing the interconnect capabilities for example limiting the connection at level of rows and/or the neighbors logic elements [53, 55]. This implies an undeniable reduction in term of flexibility, paid to the need of guarantee small area budget. On the programming side, the increase of design constraints and then the reduction of degrees of freedom in the mapping of algorithms imply that any inefficiency of the automated high-level synthesizer leads to a dramatic loss in terms of per-

formance. To avoid this, many reconfigurable devices provide structural languages in which operators are directly mapped into the device without synthesis and the application designer can tune, refine or re-write from scratch the implementation in order to maximize the performance benefit in the same way that a DSP programmer could use the assembly language. All these preliminary considerations can be summarized in few points that we can see as requirements for an application development environment in the field of reconfigurable computing:

- to be appealing for the wide world of software and DSP programmers, such environment needs to be as similar as possible to traditional software-only environments.

- to be effective and compliant to the huge investment in term of area and costs required by reconfigurable hardware, such environment needs to provide capability to exploit as much as possible architectural features.

3.3 Instruction set extension implementation on a standard compilation tool-chain

The extension of a standard software tool-chain in order to support instruction set metamorphosis implies to analyze the role played by each tool and the efficiency required by each step, with the final goal of proposing to application developers a tool-chain in which hardware and software can be handled together. The introduction of instruction set extensions implies modifications in each step of the compilation process, and the addition of configuration tools dedicated to the mapping of instruction set extensions in the reconfigurable hardware. In this section, we focus on the software supports necessary to handle instruction set reconfiguration from a C compiler, while aspects concerning the extension definition and its mapping on the hardware support, will be dealt with in the next sections.

In general terms, modifications in the assembler and in the linker are kept as minimal as possible, since the assembler can be reduced to a simple mnemonic translator and the linker needs to include the eventual bitstream for the hardware customization. On the contrary, the high-level compiler needs to be conscious of the reconfigurable parts in order to help the user in the optimization process. We can require to programming tools for reconfigurable processors many tasks:

- to provide to the user the capability to define and instance an extended instruction;
- to schedule the extended instruction accurately;
- to automatically recognize user-defined extended instructions in a general-purpose code;
- to detect critical kernels and automatically generate a set of extended instructions.

The definition of extended instructions is usually accomplished by dedicated tools, while the capability of instancing the extended instructions in a software code can be obtained by using the same functionalities provided for assembly inlining. The last three points are very specific of reconfigurable computing. Accurate scheduling and identification of custom instructions can be handled by modifying the machine description and the intermediate representation (a sort of virtual machine-independent assembler) of the compiler. In the case of traditional C tool-chains (e.g. GNU GCC [97]), this implies the complete recompilation of the compiler front-end since the machine description is static. It is thus possible to deal with extended instruction in the same way that a compiler handles floating point extension, describing the required functional units in the intermediate representation [98]. Of course, this proves to be a hard obstacle for most of the application developers, also in terms of time required during the design-space exploration when the instruction set extension is under definition.

Alternative approaches have been proposed in research projects on advanced high-level compilers like Impact [21] and SUIF [22]. In this cases machine descriptions and intermediate representations can be dynamically extended without rebuilding the tools, since the description of the target architecture is read before each compilation. This implies that all the internal automata required to implementing, for example, the pattern matching and the scheduling steps are dynamically generated from the architecture description. A state of the art compiler able to handle the optimized scheduling of custom instructions (even if featuring long latencies) can be found, for example, in the MOLEN project [12] or in the DRESC framework [13], respectively based on SUIF and Impact. Moreover, the Trimaran framework [14] proposes a scheduling mechanism based on simulation and profiling back-annotations to reduce the stalls in an execution-aware environment, although the impact on the compilation time.

This point introduces the last issue that reconfigurable computing imposes on programming tools that is the reconfiguration of the simulator. Similarly to the compiler, the simulator needs to be adapted to the changes or the extensions of the instruction set. Language for Instruction Set Architecture (LISA), commercially available from CoWare and Axys, as well as open-source architecture description languages, like Arch-C, are examples of frameworks where cycle-accurate instruction set simulators can be built with the support of a native structure implementing typical processor objects, like the pipeline or the register file. This approach requires to rebuild the instruction set simulator every time the instruction set is changed. In [15] an alternative approach is proposed. A dynamically linked library is used to model the instruction set extension, while the main processor is modelled by standard simulator support. The mechanism, described in the following of this thesis, is applied on both functional and cycle-accurate simulation, integrating the mechanism on an environment based on LISA and SystemC, and on a pure-functional debugging environment based on the GNU GDB simulator.

Figure 3.2 shows a simplified and very general block diagram for a programming environment supporting reconfigurable computing. It includes

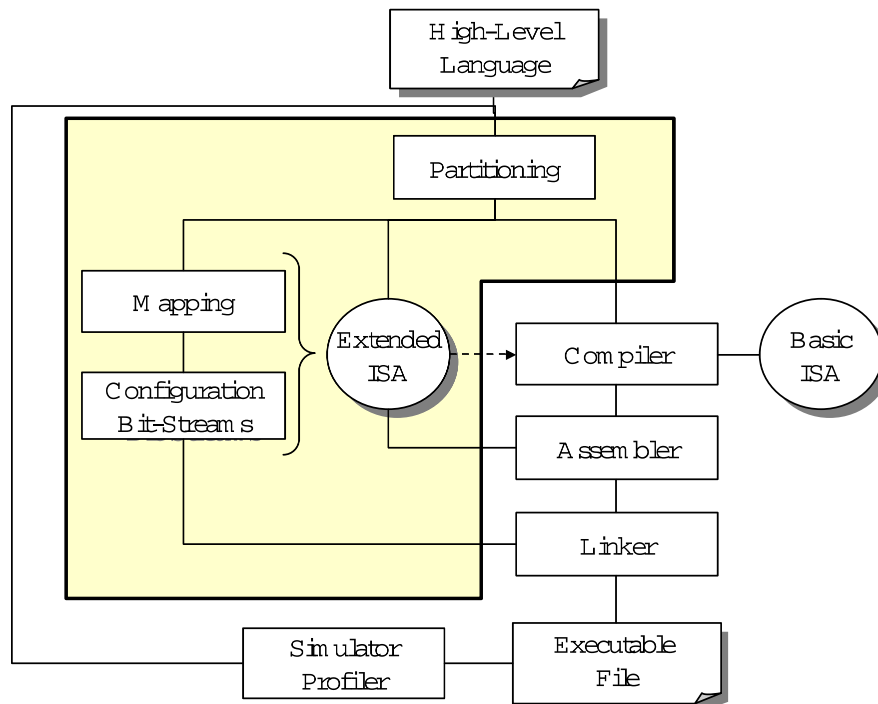


Figure 3.2: Basic software tool-chain extension to support reconfigurability issues

the basic software support (compiler + assembler + linker + simulator) previously described, and the partitioning and configuration parts. The partitioning is the process, automatic or not, of design-space exploration in which critical tasks or kernels are moved from a software implementation to a hardware one (and viceversa) depending on the required performance (speed, energy, ...). Today, this process is usually under the whole control of the programmer, although it can be helped by the usage of tools. Many researches are going in the direction of full automation of the partitioning since it will represent the most appealing enabling step toward the true soft-programmable hardware (e.g. [6]). Despite this, very few works are leaving the academic/research project to challenge the market, and these few works are focused in the field of mask-time programmable device (e.g. [56]). Even if constrained to provide good area figures to be appealing for the integration on system-on-chips, reconfigurable devices

remain very precious resources (and area demanding) that shall be return by high performance. The programming efficiency required for run-time dynamically reconfigurable devices can be accomplished only by the full exploitation of the computational capabilities of that, with a very restricted margin for the overhead that an automatic design flow can introduce. This is probably the most important difference between configurable solutions (like mask programmable, application specific standard processor, and so on) and reconfigurable solutions, that heavily impacts in term of programming models and languages. In fact, while for configurable solutions the literature as well as commercial proposals are talking of high level description languages, like the C, the common proposal for dynamically reconfigurable devices is some kind of structural form, like the assembler, as will be described in the following of this chapter.

The last block in Figure 3.2 is the configuration engine, a tool that starting from some kind of description language is capable of providing the configuration bitstream for the reconfigurable device. This tool is (of course) tightly coupled with the underlying hardware, and for C-based configuration flow it represents the bridge from the software to the hardware worlds.

3.4 Bridging the gap from hardware to software through C-described Data Flow Graphs

Programming of reconfigurable devices can be performed in many different ways, borrowing methods and tools from standard hardware design (VHDL or Verilog) or borrowing methods and tools from software compilation. As stated in [45], there is no a real difference from high-level behavioral synthesis and non-optimizing compilation of programming languages, since they are basically translations of the initial language to an intermediate representation. On the contrary, the optimization is a very different step in hardware synthesis from the software synthesis, with different metrics and cost-functions. Another common point between compi-

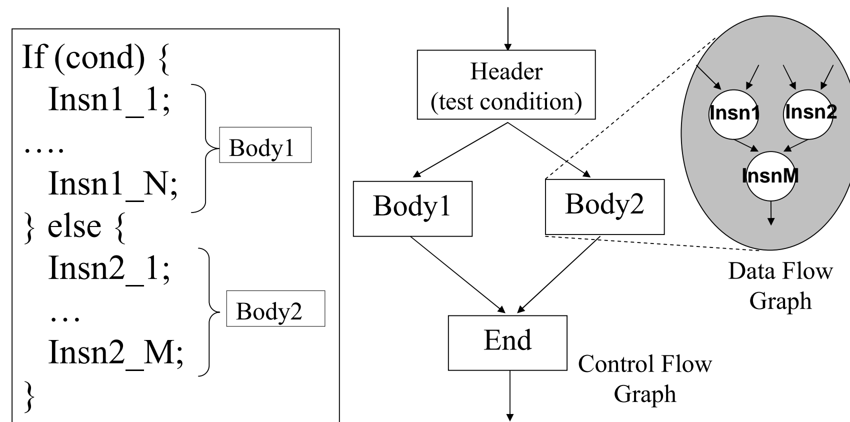


Figure 3.3: Examples of control and data flow graphs

lation and synthesis is that graphs are most often used for internal representations. In software programs, we can distinguish between two kinds of graphs: Control- and Data-Flow Graphs (respectively CFG and DFG). The CFG is the representation of the paths that might be traversed through a program during its execution. Each node of the CFG is known as basic block and its graph representation is a DFG. The DFG describes the dependencies among the set of operations required to the data processing. As shown in the example in Figure 3.3, branches of a conditional statement (if…then…else …) are represented as nodes of the CFG, while the operations performed in each branch are described by a DFG "attached" to a CFG node.

In hardware description languages there is the co-existence of both sequential and concurrent definitions of operations. As an example, the behavior of a process or the expression assigned to a signal follow a sequential paradigm, although this not means that the same semantics of the software languages, like C, are used. They can be viewed as nodes in the DFG, as well as sub-graphs of a DFG, depending on the granularity we assign to the node. In any cases, hardware description languages use a event-driven activation mechanism in which more than one DFG and more than one DFG node can be active per time natively, and this point

represents the most significant difference with respect to control parts of software languages. Of course, during the compilation for processors featuring some degree of parallelism (e.g. VLIWs, Superscalars, TTAs, ...) this constraint is heavily relaxed, bringing software implementation near to the hardware, although the different optimization metrics.

For the definition of a suitable bridge between hardware and software in the field of reconfigurable computing, the DFG probably represents the most natural choice. In fact, for reconfigurable processors the control part is typically managed by the processor core, while the hardware acceleration is provided for the DFGs. Hence, the DFG suitable for the mapping on the reconfigurable device can be described by a sequential language, like C, but it can be viewed at the same time as an abstract representation of a circuit.

The exploitation of the parallelism is the key point for the effectiveness of the reconfigurable computing, be it at word-level or loop-level. Standard software compilation techniques like software pipelining [61], iterative modulo scheduling [62] and vectorization [60] are examples of well-known methods that increase the instruction-level parallelism by the exploitation of loop-level data parallelism. Loop transformations are widely used in compilation for VLIW processors to maximize the performance, as well as they are applied to utilize SIMD (Single- Instruction Multiple-Data) extensions (like the Intel MMX or AMD 3DNow!). These methodologies can be efficiently applied in order to transfer loop-level parallelism to the instructions in the loop body, thus increasing the instruction level parallelism of the innermost DFG, while more hardware-oriented methods can be applied for the efficient mapping of DFGs on the reconfigurable devices. Starting from a DFG software description where the instructions (or DFG nodes) are executed in the same order in which they are written in the code, we can relax the enabling rule of the DFG executing each node when inputs are available and output can be overwritten, as described in [63]. The run-time execution of a DFG can thus be modelled by Petri Nets as in [64, 65]. Furthermore, by nodes scheduling and registers insertion it is possible to build the DFG in a pipelined form, without affecting the

functionality. In this case, it is possible to overlap the execution of successive DFG activations (if the data dependencies allow that) hence improving the performance by the exploitation of parallelism at level of iteration, as in [59, 60].

To this point, we have discussed about the role played by DFGs as bridge between software and hardware. One more point is of course represented by the way in which the DFG can be described in order to meet the requirements of effectiveness and friendliness posed as basis of a programming tool-chain for reconfigurable processors. We said that the entry language must be appealing to software programmers and must be effective in term of hardware utilization. An interesting option is to use the C language for that goal: it allows to describe DFGs since DFGs are representations of the basic blocks, but it also allows to handle the DFG topology under simple restrictions. For example, the utilization of a single-assignment form, in which each variable is assigned exactly once, can help the user in the DFG modelling, providing a simple way of handling efficiently all the data dependencies, as proposed in [40]. The single-assignment form is today introduced in many compilation frameworks as an important intermediate representation in order to both simplify and optimize the internal compilation steps. Starting from the version 4, also the GNU GCC makes extensive use of single-assignment representations although the conversion to single assignment is performed (in my knowledge) only for scalar register values (everything except memory) at level of basic block. Therefore, conditional statements (if...then...else) are converted in a speculative form which executes concurrently each branch of the statement and introduces *merge* nodes that select the correct outputs among the branch-replications, similarly to functionality provided by multiplexers in the hardware design.

In general talking, the translation of standard C and C-dialects into some kind of hardware description is a complex problem addressed by many research programs [36, 37, 38, 5, 39], especially if we include the memory access (i.e. pointers) [43]. In the case of reconfigurable processors, the processor core can handle (and usually handles) the memory access,

eventually with the help of DMAs to speed-up the memory access, thus simplifying the synthesis requirements. Summarizing, single-assignment forms are a restriction of the C semantic, they are useful to accurately handle the DFG performance (parallelism and pipeline structure) and they can be extracted from high-level C compilers. The application developer can thus start the implementation over a reconfigurable processor from the application description written in C, and selecting the critical kernels suitable for the reconfigurable hardware mapping. Depending on the efficiency required, the application developer can choose to use an automatic translation mechanism or to hand-code the kernel with a low-level description language, thus introducing a third trade-off point represented by the time spent to the development.

3.5 Overview of programming tools for reconfigurable processors

Programming frameworks for reconfigurable architectures are highly dependent on the structure, the hardware granularity and the language proposed as entry-point. Although far from being an ideal hardware description language, C was selected as an appealing entry-point for the configuration of reconfigurable processors since the first architectures (e.g. PRISM [16]). Milestones of the research on field of reconfigurable processors, like the Garp [19] processor, and commercial state-of-the-art reconfigurable processors [23, 24, 25, 26] proposed C-based design environments envisioning the possibility to offer the end-user the capability of automatic partitioning, and then to co-compile the same source code over both the processor core and the reconfigurable logic. The Nimble compiler [80], targeting the Garp processor, is one of the first tools that try to automatically move critical kernels from the processor core to the reconfigurable hardware accelerator, selecting them from the basic blocks found in the innermost loops. PipeRench [46, 47], one of most popular coarse-grained reconfigurable data-paths, is configured using a single-assignment lan-

guage with C operators (called DIL, Dataflow Intermediate Language), as well as RaPiD [48] that features a C-based proprietary language. RaPiD-C programs consist of nested loops describing pipelines, and language extensions allow the programmer to explicitly handle synchronization mechanism, specify parallelism and data movement (that is stream-based). Another example of popular coarse grain architecture is represented by the RAW architecture [49] developed from the MIT: in this case a SUIF-based compiler partitions the application over a mesh of RISC processors, instead of performing a technology mapping. Another programming approach based on C language was provided for the NAPA architecture [50], including a C-programmed reconfigurable device as I/O coprocessor.

Many reconfigurable devices are programmable at assembly-level or by graphical tools (for manual mapping), in a way that seems to trade part of the programmability offered by high-level languages with the programming efficiency (MOPS/mm^2), as reported in the Hartenstein's retrospective [30]. In general, the underlying architecture has a strong impact on the technology mapping, on the placement and to a lesser term on the routing algorithm. Direct mapping is probably the most used method for coarse grain architectures, where operators are mapped to the programmable elements that compound the device without a real logic synthesis step. PACT XPP [53] and MorphoSys [55] are effective examples of this approach, although they provide a tentative to virtualize the underlying layer using C-based high-level compiler flows [60, 13]. For the full exploitation of the architecture capabilities, PACT XPP is programmed through the Native Machine Language (NML), a structural event-based netlist description language. The following code is an example of nML code.

```
MODULE LDPC_VNODE_WC_2_SINGLENODE(DIN Q0,A0,B0, DOUT OUT)
{
       OBJ q0_plus_b : ADD @ FREG 0,0 {
  A = Q0
  B = B0
       }
       OBJ q0_plus_a : ADD @ 0,0 {
  A = Q0
  B = A0
       }
```

```
        OBJ clip_a : CLIP (8) @ 0,0 {
   A = q0_plus_b.X
        }
        OBJ clip_b : CLIP (8) @ 1,0 {
   A = q0_plus_a.X
        }
        OBJ pack : PACK @ FREG 1,0 {
   A = clip_a.X
   B = clip_b.X
        }
        OUT = pack.X
}
```

In the example, sums and clipping instructions are manually placed to cells 0,0 and 1,0, whereas FREG register are used to transfer data between two successive rows. In fact, PACT XPP not features a vertical routing channel and vertical data transfers are performed only by registers in a pipelined form. Specific tools are proposed for the place-&-route phase as reported in [54], where the strongly pipelined structure requires to pipeline also the interconnections across rows by dedicated registers.

For the MorphoSys architecture, a SUIF-based compiler is provided for the host processor, while the partitioning between hardware and software is performed manually by the programmer. The MorphoASM, a structural assembly-like language, is used to configure each programmable element to the functionality required. Usually, the programmer needs to take into account also the interconnect capabilities of each programmable element in order to distribute the processing elements in the device pipeline in a way compliant to the timing requirements. An example of MorphoASM code is reported in the following (*star*s give the programmer the possibility to specify manually row and column).

```
CELL{*,*} R13 = MULSIH{FB{InputTwiddleCos, 0, OMEGA_BR2, COL_BUS,
WORD}, R5, R14} << 1;
CELL{*,*} R12 = MULSIH{FB{InputTwiddleCos, 0, OMEGA_BR2, COL_BUS,
WORD}, R1, R14} << 1;
CELL{*,*} R11 = MULSIH{FB{InputTwiddleSin, 0, OMEGA_BR2, COL_BUS,
WORD}, R1, R14} << 1;
CELL{*,*} R10 = MULSIH{FB{InputTwiddleSin, 0, OMEGA_BR2, COL_BUS,
WORD}, R5, R14} << 1;
CELL{*,*} NOP{}; CELL{*,*} R13 = ADD{R13, R11} >> 1; // scale down;
CELL{*,*} R12 = SUB{R12, R10} >> 1; // scale down;
```

```
CELL{*,*} R11 = MULL{R15,R4} >> 1;// scale down for adjustment;
CELL{*,*} R10 = MULL{R15,R0} >> 1;// scale down for adjustment;
CELL{*,*} NOP{}; CELL{*,*} R8 = ADD{R11, R13};
CELL{*,*} R9 = ADD{R10, R12};
CELL{*,*} R4 = MULSIL{R15,R4,R8}; // scale down
CELL{*,*} R0 = MULSIL{R15,R0,R9}; // scale down
```

The mapping on reconfigurable architectures, especially for coarse grain architectures which are very different from the island style of FPGAs, requires specific management constructs. As an example, in the Garp processor, the GAMA tool [79] maps a DFG using a dedicated tree covering algorithm that split the original graph into sub-trees with single fanout nodes, introducing a significant overhead in the resources utilization. Furthermore, only acyclic graphs are supported. Modules detected by the tree covering are placed in Garp array rows (only one module per row) using bit-slice methods proposed for data-paths synthesis in regular architectures. The DRESC compiler [13] is an example of high-level compiler targeting a MorphoSys-like coarse grain architecture. It focuses on the exploitation of loop-level parallelism and performs the place-&-route for the reconfigurable hardware using an extended iterative modulo scheduling algorithm. A simulated annealing strategy is used to decide when a legal configuration can be accepted or not, helping to escape from local minimum. In some cases, where the reconfigurable processor is integrating an embedded FPGA or it is implemented on a stand-alone FPGA (like in MOLEN [12]), VHDL and Verilog are used for the hardware customization: the optimization process is that one typical of hardware design on FPGAs where the behavioral HDL descriptions are substituted by FPGA-specific macros, when expected performance are not achieved directly from synthesis. On the programmability side, since HDL is the entry-point, all the C-based language generating VHDL can be applied for a more software approach, but the optimization process is performed under the hardware design paradigm, for example analyzing timing constraints and critical paths.

3.6 Griffy project overview

This section introduces the Griffy project, a programming environment for reconfigurable processor focused on C language for the processor core and a simplified C syntax for the reconfigurable device. Implementation details of the most significant steps will be described in the following chapters and results achieved developing applications will be provided. The approach has been originally applied to the XiRisc reconfigurable processor [66], developed at the Arces/STM joint lab of the University of Bologna, and it is currently applied also to the DREAM adaptive DSP [75] and to the XiSystem [67] integrating in the same chip XiRisc and a standard eFPGA. All these systems are based on processor cores driving one or more reconfigurable devices (connected as functional units or co-processors), thus providing the system reconfigurability at level of assembly instruction. As an example, in the case of XiRisc, the reconfigurable device (the PiCoGA) is fit in the processor pipeline as an additional functional unit, triggered by a dedicated assembly instruction called `pgaop`, whereas in the DREAM architecture the reconfigurable co-processing subsystem based on PiCoGA-III is handled, for simplicity, by standard memory access (as an example, computation is triggered by store operation). The functionality associated to an extended instruction (in the following commonly called `pgaop`) is modelled as a DFG, implemented in a pipelined form in order to enhance computation performance.

Figure 3.4 shows the overall programming environment proposed to the application developer. Through the profiling analysis the programmer can identify the critical kernels suitable for the mapping on reconfigurable hardware, evaluating the performance of the new implementation in order to find the best partitioning between reconfigurable hardware and software. The compiler tool-chain is based on a retargeted version of the GNU GCC and instruction set extensions are handled through assembler inlining. No specific scheduling support is provided for the extended instruction set, although recent versions of GCC (3.2 and later) counts the number of semicolons (";") inside the assembler inlining tem-

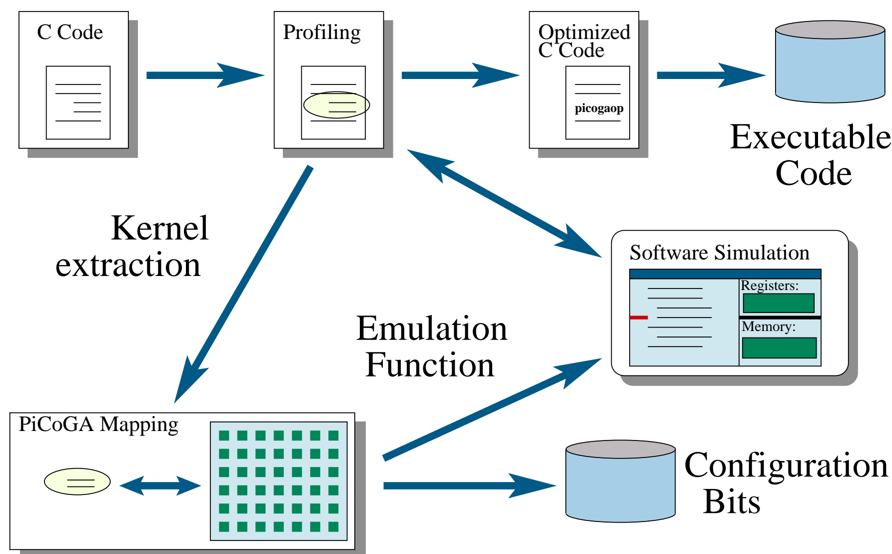


Figure 3.4: Griffy Algorithm Development Environment

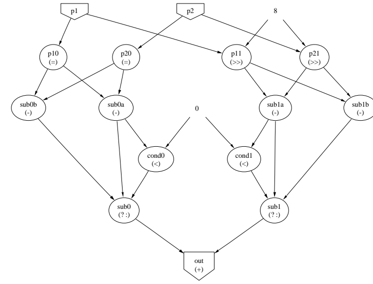
plate in order to roughly estimate the number of cycle required (the idea is to consider 1 cycle per instruction, thus 1 cycle per semicolon is considered). Software simulation is provided for both pure functional debugging on the GNU GDB and cycle-accurate instruction-set simulation with the LISA and System-C supports. Differently to the compiler, the simulation environment supports the instruction metamorphosis through the utilization of a dynamically linked shared library (.so library under Linux environment). The emulation library of the instruction set extension is automatically generated from DFG compilation, and is currently successfully plugged in both GDB and LISA/System-C environments [15].

The functionality of each instruction set extension is described starting from a single-assignment manually-dismantled C syntax called Griffy-C [71]. Griffy-C is a structural description in which basic C operators, like sum, subtraction, bitwise logical operation and comparison, are directly mapped on hardware resources, without logic synthesis. Figure 3.5(a) shows an example of the Griffy-C code used to implement a simple sum of absolute differences (SAD) required on video encoding applications and the corresponding (non-optimized) Data Flow Graph. Griffy-C does not support control flow statements, with the only exception of the conditional

```
#pragma pga sad4 1 2 out p1 p2
{
short int sub0a, sub1a, sub0b, sub1b;
unsigned short int sub0, sub1;
unsigned char cond0, cond1, p10, p11, p20, p21;
#pragma attrib sub0a, sub1a, sub0b, sub1b SIZE=10
#pragma attrib cond0, cond1 SIZE=1

p10=p1; p11=p1 >> 8; p20=p2; p21=p2 >> 8;
sub0a=p10-p20; sub0b=p20-p10;
cond0=sub0a<0;
sub0=cond0 ? sub0b : sub0a;
sub1a=p11-p21; sub1b=p21-p11;
cond1=sub1a<0;
sub1=cond1 ? sub1b : sub1a;
out=sub0+sub1;
}
#pragma end
```

(a) C-level description



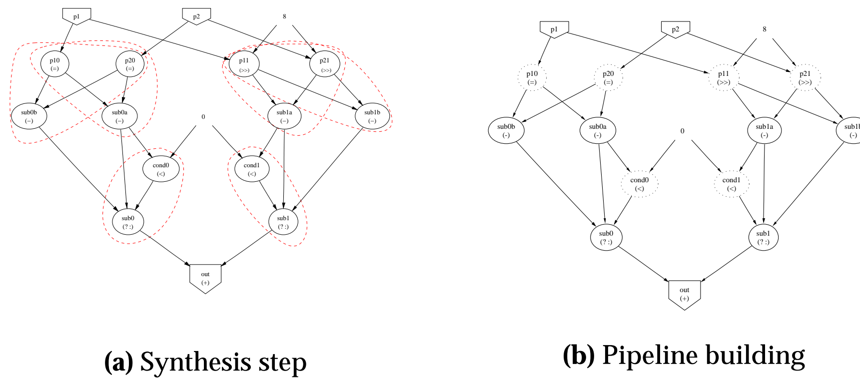
(b) DFG - Graphical view

Figure 3.5: DFG Description

assignment (“? :”) used to implement multiplexers, in hardware terms, or the merge-node under the dataflow paradigm. Detailed description of the Griffy-C syntax is provided in Appendix A.

The C-oriented description implies that some operations with constant operands may be resolved by constant folding and collapsing on following nodes. This kind of operators do not need explicit instantiation of processing elements, an this kind of optimization can be regarded as a very basic synthesis step. An example of such approach is the utilization of the routing resources to implement constant amount shifts in a fine grain routing architecture. Figure 3.6(a) shows the collapsing of shifts used in the previous SAD example for unpacking the input variable, thus providing in the Figure 3.6(b) the optimized pipelined DFG. In the figure, nodes are depicted aligned per pipeline stage and dotted nodes represent the collapsed operators.

The single-assignment syntax used in Griffy-C allows the user to handle accurately the pipeline structure and at the same time can be automatically generated from a high-level compiler tool-chain as proposed in [88].



(a) Synthesis step

(b) Pipeline building

Figure 3.6: Example of optimization of routing-only operators

Specific extensions for the definition of the variable size at bit-level are provided through *#pragma* directives in order to reduce the area occupation. The mapping process can be divided in five main steps:

- *Instruction-Level Parallelism extraction.* Starting from the data dependencies of the DFG, an optimized scheduling algorithm builds the pipeline structure. Griffy-C code is analyzed and scheduled in pipeline stages applying an earliest firing rule, in which instructions are executed as soon as possible. Detection of routing-only instructions allows to build optimized pipeline stages, although the presence of an eventual internal state (e.g. described by static variable in the C syntax) requires special management.

- *Physical Mapping.* The arithmetic and logic operations that require computational resources on the array are generated with a proper configuration. The result is a netlist, annotated with configuration bits, where elements are hierarchically organized for pipeline stage and macro-elements (the set of basic computational blocks implementing a Griffy-C operation).

- *Placement, routing and pipeline synchronization.* In this phase the netlist is arranged on hardware-specific resources, while synchronization mechanisms required for the pipeline evolution are programmed.

- *Bit-stream generation* is the last step in the configuration process, and provides the set of bits necessary for hardware configuration in a C vector form that can be included in all the standard processor toolchains.

The validation process or debugging is an additional key point required to an algorithm development environment. In reconfigurable computing handled by a processor core, the overall simulation can be managed by a standard software debugger like that one provided in the GNU environment by GDB (and its graphical interface DDD) or by the cycle accurate tools provided for LISA/System-C environments. In both cases, in the Griffy-C approach the validation of the reconfigurable part is handled by a separate viewer that shows the same Griffy-C code written by the user annotated with intermediate results. Breakpoints and control flow management are in general handled by the processor debugger, and the application developer can only inspect the status of the reconfigurable unit. As an example, Figure 3.7 provides a screen-shot of the GDB-based debugging environment augmented with the Griffy code viewer.

Applications development under the Griffy-C approach is a process in which the programmer can *move* the application from software to hardware in a sort of continuous space. In fact, starting from the original C code, the user manually rewrites the code in Griffy-C, usually working with C-based operators and then starts the performance analysis. The partitioning between C-code on the processor and Griffy-code on the reconfigurable device is an iterative process of refinement where experience and knowledge plays a very important role. But, differently to methodologies borrowed from FPGA design, this kind of approach is mainly software-oriented. In fact the user can change the partition and can optimize the kernel mapped on the reconfigurable hardware in the same way that DSP programmers use assembly for speeding-up their applications. Optimization of Griffy-C code can be performed at two main levels: pipeline restyling and intrinsic optimization. In the first case, the pipeline structure is modified playing with data-dependencies in order to retime the graph or to adjust the write-back points in pipeline, for example using software

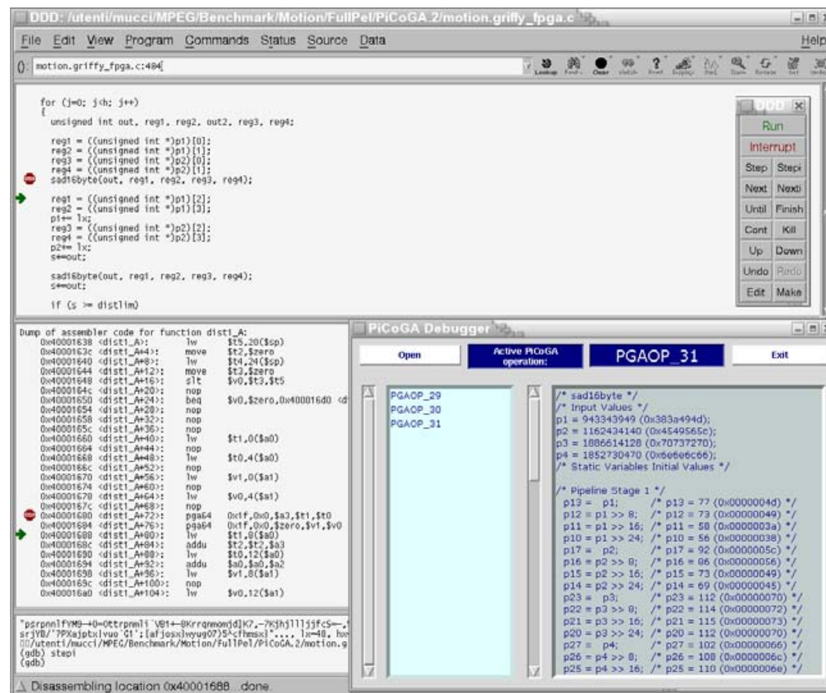


Figure 3.7: Griffy-C Debugging and Validation Environment

pipelining methods [59, 60]. In the second case (intrinsic optimization), the programmer can substitute part of the code with optimized operations like the direct instance of a look-up table. For software programmers this seems the assembly-level optimization in which high-level code is substituted by built-in functions, linear assembler or assembler, since the syntax remains strongly sequential and imperative, without any kind of direct parallelism exposition. Only tools are responsible of that.

Chapter 4

Mapping DFG on reconfigurable devices

This chapter describes the mapping of Data Flow Graphs (DFGs) described by Griffy-C on a reconfigurable device. DFGs are implemented in a pipelined form, in order to improve the final performance. Instructions scheduling is required to transform software DFG into a pipelined DFG, although some optimization steps can be applied only under some hypothesis on the underlying architecture. After a general part, the chapter includes the description of target-specific back-end flows for the mapping on PiCoGA (in particular, for the 3rd release) and on a commercially available eFPGA, through the generation of VHDL description.

4.1 ILP exploitation through pipelined DFG and Petri Nets

Griffy-C code, as described in Appendix A, features a single-assignment manually-dismantled syntax in which each operation is described by:

$$dest = function(src1, src2, \ldots, srcN);$$

Single-assignment form means that each variable can be assigned only once, while the manually-dismantling underlines the fact that each $function$

is defined by a single operator. Griffy-C syntax borrows most of the operators from the C syntax (in fact, Griffy-C is a simplified C syntax), but also includes a set of built-in functions (or hard-macros) useful to instance optimized, and commonly target-specific, functionalities (similarly to built-in functions or intrinsics of DSPs). As an example, the capability to directly specify LUTs on the PiCoGA is offered by means of a dedicated hard-macro. The semantic of Griffy-C is strongly sequential, as in ANSI C, and no parallel statements are defined. This means that parallelism is extracted from the code.

Given a set of instructions I, a sequential semantic rule implies to execute instructions in the same order they are defined. This firing rule defines the data dependencies among the different instructions. Under the Von Neumann paradigm, that is the underlying paradigm of software programming, data are stored in the memory and the access to the memory defines a sort of synchronization point. Since variables are implemented as memory locations, the access to these memory locations can be used to define the concept of data dependency. There are three types of data dependencies, which also happen to be the three data hazards:

- Read after Write (RAW or "True"): I_1 writes a value used later by I_2. I_1 must come first, or I_2 will read the old value instead of the new.
- Write after Read (WAR or "Anti"): I_1 reads a location that is later overwritten by I_2. I_1 must come first, or it will read the new value instead of the old.
- Write after Write (WAW or "Output"): Two instructions both write the same location. They must occur in their original order.

Due to the single-assignment form of Griffy-C the third kind of dependency cannot occur, while RAW and WAR can occur. If only RAW dependencies occur and then the resulting graph will be a *direct acyclic graph*. Variables read before written implement an internal state, and are implemented in C using the static attribute. Exploitation of the available instruction level parallelism needs to relax the sequential semantic rule

preserving the behaviour of the block, thus preserving the data dependencies.

Let us consider the set of instructions I defined as:

$$I = \bigcup I_i$$
$$I_i =< operator, dest_i, \bigcup_j src_{i,j} >$$

where the index i defines the order in which instructions are declared, thus executed. Moreover, let us suppose that at a given time t the set of variables $vars(t)$ be available. Then the instructions that can be executed at time $t + 1$ are that ones for which is true:

$$\bigcup_j src_{i,j} \subseteq vars(t)$$
$$dest_i \cap src_{w,z} = \phi \qquad \forall w < i, z$$

While the first check preserves the RAW dependencies, the second one is required to preserve WAR dependencies and it is always verified for direct acyclic graphs. Substituting the fully sequential semantic rule with this set of relaxed rules, it is possible to execute concurrently a set of instructions, based only on the verification of the data dependencies. Data Flow Graph (DFG) representation in Fig. 4.1 shows the result of this instruction reorganization (or scheduling), where instructions (represented as nodes) are aligned for execution time. Edges represent the data dependencies among nodes: forward edges represent RAW dependencies, while backward edges represent WAR dependencies. This kind of scheduling is also known as As-Soon-As-Possible (ASAP) scheduling policy, since instructions are executed in the first *safe* temporal slot.

Under the Von Neumann paradigm, a central memory stores the variables and a processing unit perform the elaboration. Storage and processing are distinct units, and the communication between them is often a significant bottleneck. On the contrary, hardware implementation joins storage and processing in the same unit. Each processing element is directly connected with the other processing elements providing the input data, thus requiring storage element to provide temporal disambiguation and to avoid Write-after-Read hazards.

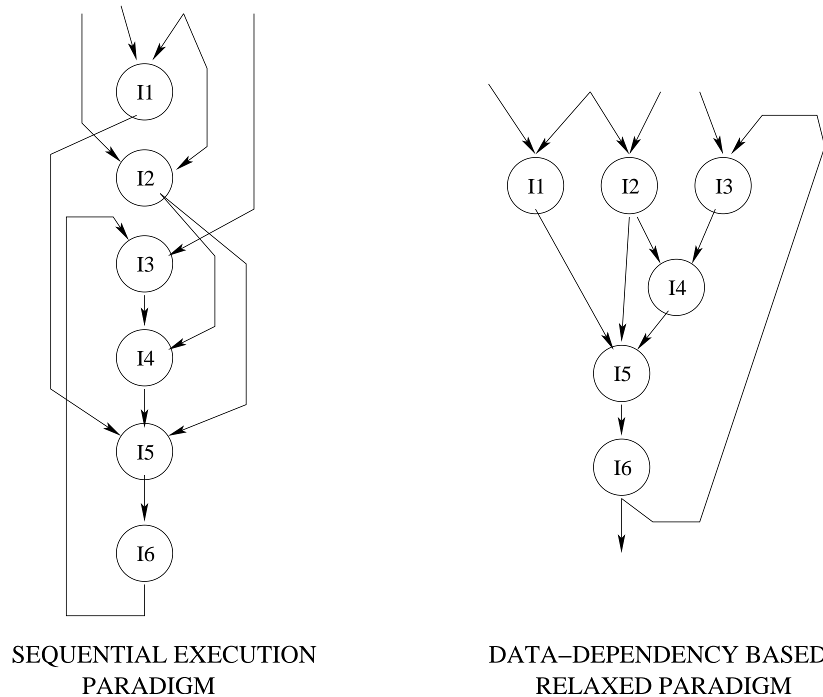


Figure 4.1: Computation paradigm relaxation preserving the data dependencies

In order to improve the computational efficiency, each DFG computation can start before the completion of the previous ones, thus overlapping (pipelining) the execution of more DFGs. To achieve this, data dependencies shall be checked in order to preserve the behaviour, verifying the possibility to update a given variable. In general terms, the computation of each node can thus start when inputs are available and outputs can be updated. This second condition is a little more complex with respect to the pure WAR safety and includes the computation time required to read and elaborate each data. For synchronous digital circuits, time is measured in term of clock cycles, although the clock cycle period depend on the combinatorial logic used. In this context, it will be supposed that the clock period is defined by the most complex computational nodes, then each computation node (or Griffy operation) requires at the most one clock cycle. Under this assumption, at a given time t the execution of each node (or instruction I_i) can be triggered when:

- inputs are available

$$\bigcup_j src_{i,j}(t) \subseteq vars(t)$$

- outputs can be updated, since
  - all the preceding nodes $I_{w<i}$ requiring the old value of I_i are already triggered, hence the old value of I_i is not more required (WAR check)

$$dest_i(t) \cap src_{w,z}(t) = \phi \qquad \forall w < i, z$$

  - given at the time t the on-going n^{th} DFG iteration, a node relative to the $n + m^{th}$ iteration can be triggered if
    * RAW and WAR dependencies for the $n + m^{th}$ iteration are verified (as in previous items)

$$\bigcup_j src_{i,j}(t, n+m) \subseteq vars(t, n+m) \quad \forall\, m > 0$$
$$dest_i(t, n+m) \cap src_{w,z}(t, n+m) = \phi \quad \forall\, m > 0;\ \forall w < i, z$$

    * all the successive nodes $I_{w>i}$ relative to the previous DFG iterations $n + k^{th}$ with $k < m$ have already read the output $I_i(t, k)$ corresponding to the $k^{th}-$iteration (temporal dependent RAWs and WARs)

$$dest_i(t, n+k) \notin src_{w,z}(t, n+k) \quad \forall k < m;\ \forall w > i, z$$

This process can be better and more intuitively modelled by Petri Nets [65]. A Petri Net is a three-tupla (P,T,A) where:

- P is a non-empty set of place denoted by $\{p_1, p_2, ..., p_n\}$;
- T is a non-empty set of transitions denoted by $\{t_1, t_2, ..., t_m\}$;
- A is a non-empty set of directed arcs;

such that $P \neq \phi$, $T \neq \phi$ and $P \cap T = \phi$, $A \subseteq P \times T \cup T \times P$. Pictorially, P, T and A are respectively represented by circles, bars and directed arcs. Each

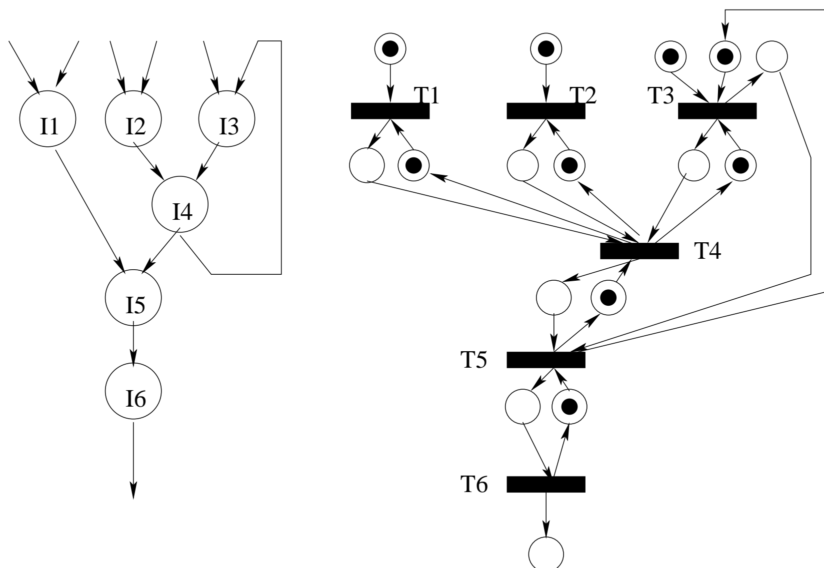


Figure 4.2: DFG and the corresponding Petri Net representation

transition is *enabled* when all the places connected to the transition have at least one *token*. In our case, we consider a subset of Petri-Net in which at most one token can reside in each place and the status is updated at discrete steps, hence each transition is *fired* when all the tokens are available (*earliest firing rule*) at discrete step of time (also known as timed Petri Net [64]).

Computational nodes are associated to transitions, and firing a transition means executing an operation. Starting from a Data Flow Graph, each edge is substituted by two arcs, respectively forward and backward (with respect to the direction of the original DFG edge). Forward arcs determine the availability of a new input data, while the backward ones determine the request of new data, under a producer-consumer mechanism. Fig. 4.2 shows an example of DFG and the corresponding Petri Net representation. Tokens, depicted as black circles, identify the initial state in which all the operations require data to compute, and (let us suppose) the primary inputs are available. Transitions featuring a token for each input arcs are T_1, T_2 and T_3, then can be fired at time t. At time $t+1$, only T_4 can be triggered, while at $t+2$ either T_1, T_2 and T_5. At time $t+3$, T_6 and T_3 can be triggered, and so on, as described in Fig. 4.3.

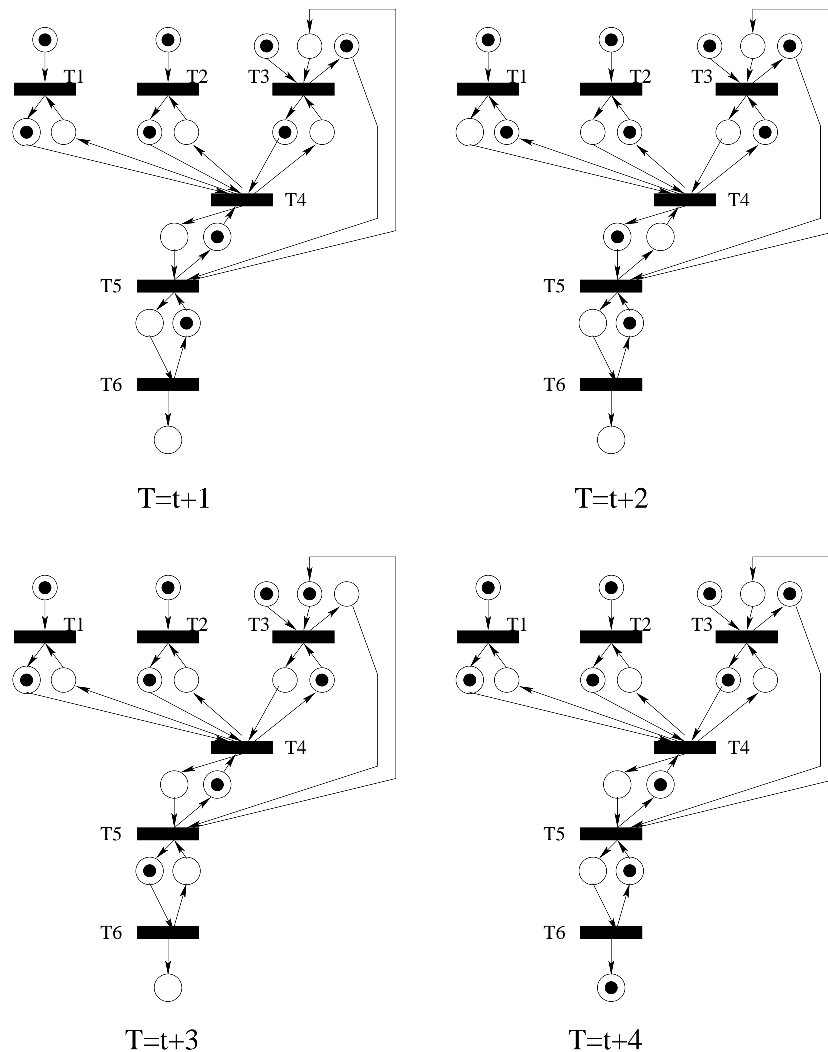


Figure 4.3: Petri Net transition firing

Pipelined execution of DFGs, under a Petri Net paradigm, implies that intermediate results are stored in registers, since temporally different instances of the same DFG are overlapped. But, also in this case, two considerations can drive further optimizations:

- in digital synchronous design, registers sample the input at the rising (or falling) edge of the clock, and their outputs don't change during the clock period. This means that the backward arcs can provide their tokens at the beginning of the clock cycle in which transitions are fired, allowing to implement more compact pipelines.

- if the target architecture features programmable routing, some operations can be implemented only by routing resources, like in the case of a shifts with constant amount. Furthermore, under some hypothesis (discussed in the following), some operation can be collapsed in the successive operations. This is the case of bit-wise logic operation involving a constant that can be reduced to selective connections (some bit connected, some bit constant to 1 or 0) or bit-wise not. If the architecture provides LUTs or input inversion logic, also this kind of operation can be considered *routing-only* since its implementation does not require specific computational resources. This optimization can be used to optimize the instruction scheduling in order to implement more compact pipelines.

Last two items improve the timed Petri Net evolution in order to achieve better pipelines and then better throughput. This is the computational schema proposed in the Griffy project, that will be described in the following, and that can be summarized in:

- a Pipelined Data Flow Graph is built exploiting the instruction-level parallelism from the sequential Griffy-C code.
- for a given DFG, RAW and WAR dependencies are preserved at compilation time by the Griffy scheduler, while hazards across different iterations of the same DFG are handled by dedicated hardware at execution time. For that:
  - without loss of generality, each pipeline stage (composed by a set of concurrent computational nodes) is considered as a Petri Net transition, under an ASAP scheduling policy.
  - pipeline management is handled by a programmable control unit, generated by Griffy tools. Each element of the control unit represents a programmable Petri Net transition which enables the execution of the respective (set of) computational node(s) depending on sources and resources availability.

It should be noted that spatial computation is preserved by the compiler, during the pipeline organization. On the contrary, temporal dependent hazards are checked at execution time, since the pipelined and overlapped execution of successive DFG iterations is dependent on conditions non-predictable at compile time, as the inputs availability, external condition of stalls and, of course, the frequency of DFG triggering.

4.2 Instruction scheduling: optimized DFG for pipelined computation

Instruction scheduling is the phase in which sequential Griffy-C code is selected to be executed in a specific pipeline stage, thus translating a sequential description in a concurrent pipelined form. This process borrows the instruction firing mechanism of the Petri Nets, since instructions can be executed in a specific pipeline stage under the same hypothesis that enable the transition firing. This section describes the selection algorithm used in the Griffy flow, starting from a simplified version for direct acyclic graphs (DAGs) and thus improving the algorithm to support functionalities holding an internal state. In the first case only RAW dependencies are considered, while in the second one also WAR dependencies will be taken into account.

After the scheduling phase, the flow becomes mainly target specific and includes for example the place & route phases. The next section will provide an overview of two target specific back-end, the first one for the PiCoGA-III (featuring a dedicated control unit) and the second one for a commercially available eFPGA programmed generating standard VHDL description.

4.2.1 Scheduling of direct acyclic graphs

A direct acyclic graph is a graph with one-way edges containing no cycles. This means that if there is a route from node A to node B then there is no

way back. In "software" words, this means that each variables is written before read, then only RAW data dependencies can happen, without static variables.

Under the *earliest firing rule*, each instructions is executed as soon as possible, thus in the first pipeline stage in which input data are available. Routing only operations can be considered as operations featuring zero-time execution, then they provides data available in the same stage in which the instruction is executed. On the contrary, the other instructions will provide variables available only in the next pipeline stage. In this case, the corresponding ASAP scheduler is very simple, and can be represented by the pseudo-code in Fig. 4.4.

$$
\begin{array}{l}
I = \bigcup_i I_i; \quad // \; Set\; of\; instructions \\
P = \phi; \quad // \; Selected\; instructions\; for\; the\; current\; pipeline\; stage \\
PS = 1; \quad // \; Current\; pipeline\; stage \\
AvailableVars = \bigcup primary\; inputs \\
while\; (I \neq \phi)\; \{ \\
\quad foreach\; I_i\; in\; I\; do \\
\quad\quad if\; (sources(I_i) \subseteq AvailableVars)\; \{ \\
\quad\quad\quad P = P \oplus I_i \\
\quad\quad\quad if\; (I_i\; is\; routing\; only)\; \{ \\
\quad\quad\quad\quad AvailableVars = AvailableVars \oplus destination(I_i); \\
\quad\quad\quad \} \\
\quad\quad \} \\
\quad end \\
\quad PipelineStage(PS) = P; \quad //Build\; pipeline\; stage \\
\quad I = I \ominus P; \quad // \; Remove\; executed\; instructions \\
\quad foreach\; P_i\; in\; P\; do \quad //Update\; available\; variables \\
\quad\quad AvailableVars = AvailableVars \oplus destinations(P_i); \\
\quad end \\
\quad PS++; \quad //Update\; current\; pipeline\; stage \\
\quad P = \phi; \quad //Reset\; selected\; instructions \\
\}
\end{array}
$$

Figure 4.4: DAG scheduling pseudo-code (with routing only optimization)

It is important to observe that while the scheduling algorithm preserves the read-after-write data dependencies inside a specific Griffy operation, in case of pipelined (overlapped) execution of successive instances of the same Griffy operation also write-after-read hazards shall be checked. In fact, each pipeline stage can accept a new computation when the out-

puts can be updated (there is a write-after-read check), under the computational paradigm explained in the previous section. This second check is provided by a Petri-Net based hardware pipeline manager that, depending on the data dependencies, allows or not to trigger pipeline stages.

4.2.2 Scheduling of data flow graphs

To consider complete DFGs means to take into account DFGs featuring an internal state. In particular two additional effects are to be considered. The first one is given by the presence of static variables (used to implement a state) for which the old value is available at the beginning of the DFG computation, be it an initialization value or a real value referred to a past computation, and that can be update only when all the WAR dependencies are preserved. The second effect is given by the presence of routing-only operations for which the optimization process would remove the corresponding memory location. Let us consider the following example:

```
static int status;

tmp1   = status << 1;
status = in + 5;
out    = tmp1 + 1;
```

In this case, standing the routing-only optimization of $tmp1$, the out variable must be considered dependent from the old value of $status$, and the scheduling algorithm shall preserve this behaviour adding a firing rules. In this case WAR hazards are verified by:

$$dest_i(t) \cap src_{w,z}(t) = \phi \qquad \forall w < i, z$$
$$alias(dest_i(t)) \cap src_{w,z}(t) = \phi \qquad \forall w, z$$

where the function $alias$[1] represents any possible direct or indirect dependency to the static variable $dest_i$. A direct dependency represents the case in which a routing-only instruction involves a static variable. Let us define this case as direct static alias, and let us define as indirect static alias

[1]the term alias is used since the operation can be considered as an alias representation of the memory location of the original variable

(and consequently a indirect dependency to a static variable) a routing-only instruction which involves a direct static alias or another indirect static alias. Under this assumption, a static alias (both direct and indirect) depends to one or more instructions holding a state. Then, the static variables shall be updated only when both direct and indirect (by means of routing-only propagation) WAR dependencies are preserved.

Let us consider now the following code:

```
static int status;

tmp1   = status << 1;
status = in + 5;
out    = tmp1 + status;
```

In this case, we have that out reads both the old and the new value of the static variable $status$. This implies that $tmp1$ cannot be optimized, since only different storage elements can preserve the original behaviour. Hence routing-only optimization depends from the overall DFG and the adopted scheduling policy, and not only from the specific instruction.

The scheduling algorithm proposed in the Griffy project supports both static variables management and routing-only optimization. ***Routing-only*** operations are detected depending on the features of the single instruction, checking the operation type and the sources involved. The PIPEREG attribute (see Appendix A) can be used by the programmer in order to force a routing-only instruction to be non optimized (for example, in order to build a delay line or to retime a graph). The scheduling algorithm tries to execute all the operations that satisfy the earliest firing rule, considering as zero-time executed all the routing-only instructions similarly to the scheduling algorithm proposed for DAG. In addition to that basic algorithm (Fig. 4.4), a check condition enables the firing of instructions involving static variables and alias of static variables.

In the first case, the computation of an instruction I_i having as destination a static variable (thus an instruction holding an internal state) is enabled only if all the instructions which need to read the old value of I_i are already executed or they are executed in the current pipeline stage. In the second case, a static alias could be triggered in the first pipeline stage

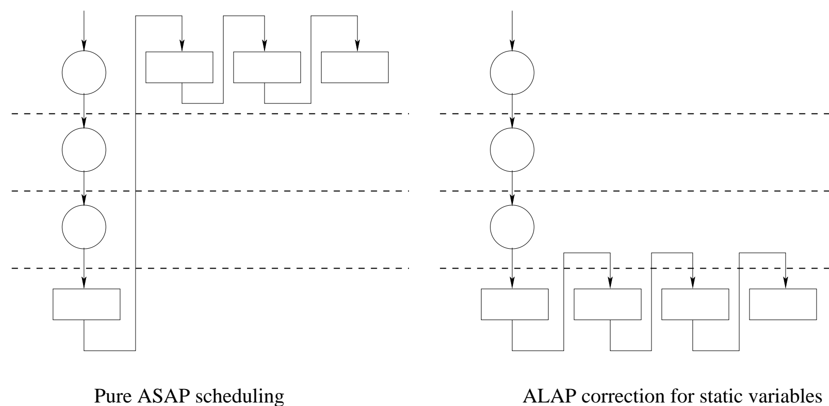


Figure 4.5: Example of ALAP correction for static variables

in which non-static variables are available (at the most, in the first pipeline stage), but this could create longer paths in the case of shift registers. As shown in Fig. 4.5, the firing of static alias following a pure ASAP policy could imply that a chain of registers is split over more than one pipeline stage, hence increasing the critical path (and the issue delay) since past values are available from the beginning. For that, the Griffy scheduling algorithm tries to delay the execution of static alias[2] As Late As Possible (ALAP), implementing in many cases a more efficient pipeline structure, as in the right part of Fig. 4.5.

On the other hand, check conditions enabling the execution of an instruction are both coherence and optimization checks. If the enabling condition is not verified, then the instruction I_i is not fired and is removed from the set of instructions suitable for the execution in the current pipeline stage. As a consequence of this *cleaning*, other instructions could be removed since I_i is not more fired and some dependencies could be not more verified.

The resulting scheduling algorithm is reported in a simplified form in Fig. 4.6, and is composed of a main loop executed until instructions are available. The loop body can be partitioned in three main sections:

- *candidate selection*, in which instructions having available sources are selected to be executed in the current pipeline stage;

[2] since the past value of a static variable can be seen as a static alias

```
I = ⋃_i I_i;   // Set of instructions
P = φ;   // Selected instructions for the current pipeline stage
PS = 1;   // Current pipeline stage
AvailableVars = (⋃ primary inputs) ∪ (⋃ static variables)
while (I ≠ φ) {
   AvailableVars = AvailableVars ∪ (⋃(static variables ∩ I))
   foreach I_i in I do // Candidate selection
     if (sources(I_i) ⊆ AvailableVars) { // RAW Check
        P = P ⊕ I_i
        if (I_i is routing only) AvailableVars = AvailableVars ⊕ destination(I_i);
        if (I_i is static) // Static written, then the old value is not more available
          AvailableVars = AvailableVars ⊖ destination(I_i);
     }
   end
   P = CandidatesAnalysis(I, P);
   if (P = φ) {
     ActivateDisambiguation(I);
   }else{
     PipelineStage(PS) = P;   //Build pipeline stage
     I = I ⊖ P;   // Remove executed instructions
     foreach P_i in P do   //Update available variables
        AvailableVars = AvailableVars ⊕ destination(P_i);
     end
     PS++;   //Update current pipeline stage
   }
   P = φ;   //Reset selected instructions
}
```

Figure 4.6: Simplified DFG scheduling algorithm

- *candidate analysis*, in which both RAW and WAR dependencies are verified, as well as the ALAP correction for static management is applied (see pseudo-code in Fig. 4.7);

- *commit stage*, in which instructions selected for the execution on the current pipeline stage (after passing the previous check) are committed and removed from list of pending instructions. Furthermore the corresponding destination variable becomes available for the next pipeline stage. In the case of no instructions selected for execution, the scheduling algorithm activates the disambiguation mode, in which conflicting routing-only instructions are de-optimized adding a PIPEREG attribute.

It should be noted, from the code in Fig. 4.7 that the ALAP correction is applied only if static alias are involved in the computation. Then, using temporary and routing-only variables it is possible to choose the way in which a set of correlated static variables is scheduled. As a choice, if the old value is passed through temporary instructions (indirect dependency) a tentative of alignment (critical path optimization) is done, while if the old value is passed referring directly the static variable, ALAP correction is not applied. It is possible to avoid this side-effect applying the ALAP correction also to static variables, but in this case the disambiguation mode shall work in two phases. In the first one, the ALAP correction will be relaxed (removed) without generating new registers, while in the second phase (the second round without fired instruction) the routing-only de-optimization shall be applied.

4.2.3 Execution-time pipeline management

Given a Data Flow Graph (DFG) organized in pipeline stages, pipeline execution of successive DFG instances shall be enabled checking at run-time the data-dependencies. For that, each pipeline stage is represented by a node of the Pipelined Data Flow Graph (PDFG) representing the data dependency across pipeline stages. Data dependencies are analyzed and for each pipelined stage is generated a pipeline controller, implementing the handshake mechanism described by the corresponding Petri-Net transition. The basic pipeline stage controller, depicted in Fig. 4.8 features:

- a *preceding* input port, providing the pipeline stage controller informations about inputs availability;
- a *successive* input port, providing the pipeline stage controller informations about the possibility to update the outputs;
- an *execution enable* that triggers the computation of the pipeline stage.

P-blocks and S-blocks are the basic sub-blocks that verify, respectively, the preceding and successive input ports. The internal structure of these

```
function CandidatesAnalysis (instruction list I, instruction list P) {
foreach I_i in I do
  written[destination(I_i)] = 0; written_in_P[destination(I_i)] = 0;
  raw[destination(I_i)] = 0; read[destination(I_i)] = 0; Read_in_P[destination(I_i)] = 0;
end
foreach I_i in I do
  foreach S_i in sources(I_i) do
    if((isstatic(S_i) = true)and(written[S_i] = 0)) read[S_i] + +; // Read past value (war)
    else raw[S_i] + +; // Read new value
    if((isstatic(S_i) = true) and (written[S_i] = 0) and (I_i ∈ P)) Read_in_P[S_i] + +;
  end
  if (isstatic(destination(I_i)) = true) written[destination(I_i)] = 1;
  if ((isstatic(destination(I_i)) = true) and (I_i ∈ P)) written_in_P[destination(I_i)] = 1;
end
do { removed = 0;
  foreach P_i in P do
    if (isstaticalias(P_i)) // Static alias check
      foreach S_i in sources(P_i) do
        if(isstatic(S_i) and written_in_P[S_i] = 0)
          RemoveReaderFromCandidate(P_i, P); removed + +;
      end
  end
  foreach P_i in P do // Static check : old value of P_i shall be yet read
    if (isstatic(destination(P_i)) and (Read_in_P[destination(P_i)] < read[destination(P_i)])
      P = P ⊖ P_i; removed + +;
  end
} while ((removed ≠ 0) and (P ≠ φ))
returnP;
}
function RemoveReaderFromCandidate (instruction V, instruction list L) {
  L = L ⊖ V;
  foreach L_i in L do
    if(destination(V) ∈ sources(L_i)) RemoveReaderFromCandidate(L_i, L);
  end
}
```

Figure 4.7: Candidates analysis algorithm

sub-blocks is reported in Fig. 4.9. Execution enables provided by the other pipeline stages are used as preceding and successive signals (directly connected or routed by a programmable interconnect), and are stored internally to the specific sub-block until the pipeline stage is fired since they can disappear after a single execution. Thanks to a feedback path, both P-block and S-block maintain the local execution enable until the global execution enable is fired, while it is reseted when after a triggering. Differ-

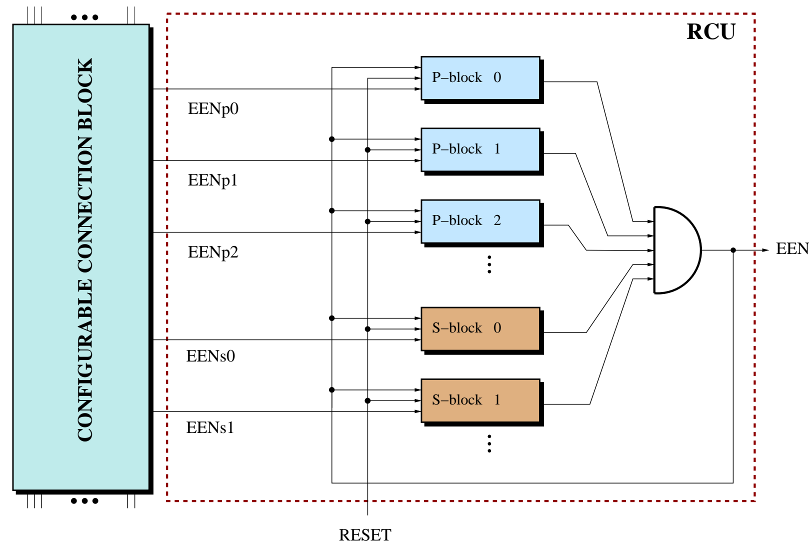


Figure 4.8: Pipeline stage controller simplified architecture

ently from the P-block, the S-block features a combinatorial path (dashed in Fig. 4.9) that allows to early evaluate the data request from successive pipeline stages, thus improving the pipeline evolution and the overall throughput (relaxing the timed Petri-Net model).

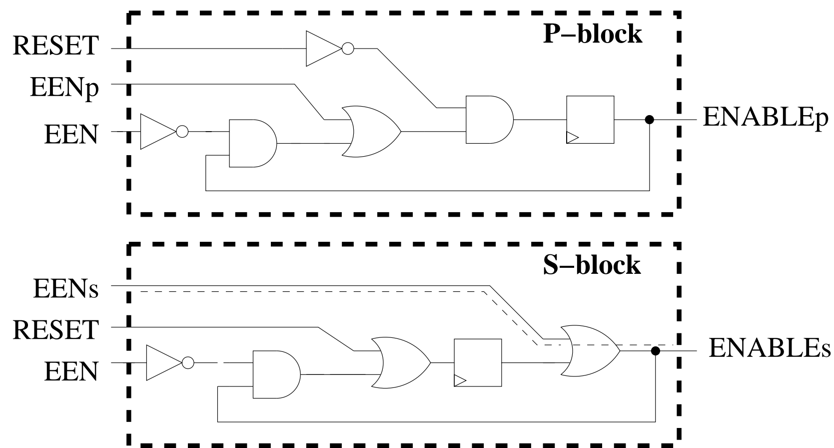


Figure 4.9: P-block and S-block simplified architecture

For each pipeline stage, the Griffy-C compiler generate a specific pipeline stage controller and the corresponding data dependencies with the other pipeline stages. In the following code, 4 pipeline stages (here termed virtual rows, or V-Rows) are described, as well as dependencies from pri-

mary inputs and outputs.

```
.output out 4

V-Row: 1
Preceding: #gins
Successive:  2
V-RowEnd

V-Row: 2
Preceding: 1
Successive:  3
V-RowEnd

V-Row: 3
Preceding:  2
Successive:  4
V-RowEnd

V-Row: 4
Preceding: 3
Successive:  #gouts0
V-RowEnd
```

4.2.4 Griffy Front-End architecture

The overall architecture of the Griffy Front-End compiler is shown in Fig. 4.10. Lexer and parser are implemented using standard tools (GNU Flex for the lexer and GNU Bison for the parser), which allow to verify the grammar of the input Griffy-C code and translate them into an abstract syntax tree (AST). The next is the (semantic) verification of the DFG description. It is performed at AST-level in order to verify the correctness of the code, including additional checks on the single-assignment form, on the variable declaration, and on the read-after-write dependency of non-static variables. When a condition is not verified, the compiler break the execution and provide an error message to the user.

Analyzing the instructions and the kind of variables involved in the computation, the compilation flow detects the instructions suitable for routing-only optimization. Then, it starts the phase of pipeline organization in which instruction level parallelism is exploited using the scheduling algorithm explained before. As a result, a netlist is generated with

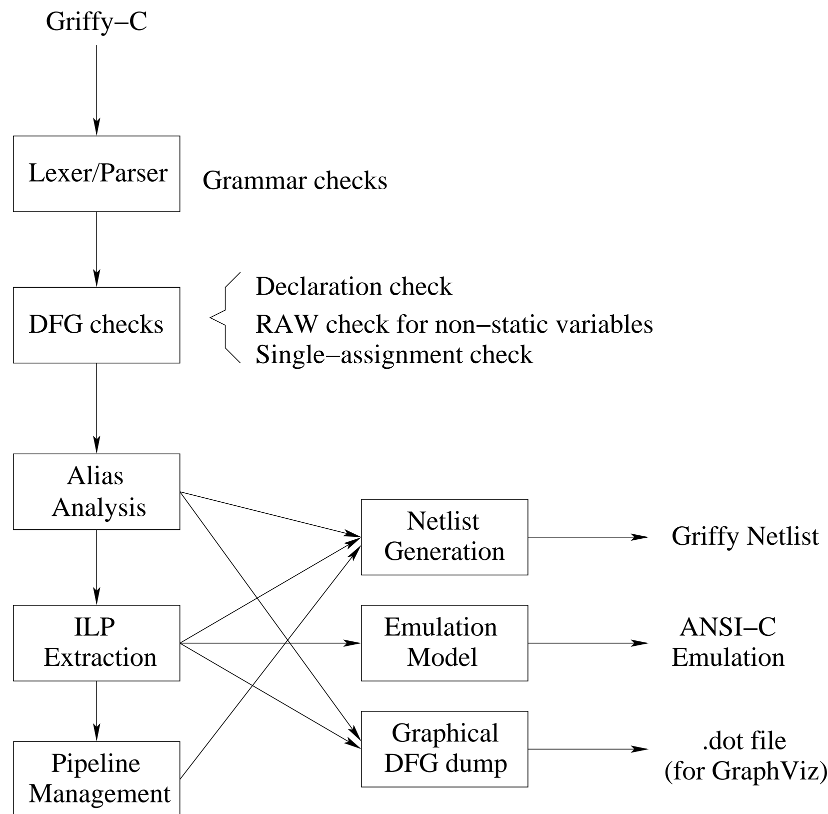


Figure 4.10: Simplified Griffy Front-End architecture

the corresponding pipeline structure and activation sequence. During the netlist generation, aliased signals (derived from the utilization of alias instructions) are substituted by the corresponding physical implementation. For example, a shift with constant amount is obtained by appropriately rearranging the input variable, and a bitwise-and with a constant is implemented by connecting the signals corresponding to 1s and by open-connection for the 0s. Furthermore, taking into account the pipeline structure, a simulation model is generated, emulating the Griffy-C code in standard ANSI-C. As a user facility, a graphical view of the pipelined DFG is dumped using the .dot format of the GraphViz tools (free download from Graph Visualization Software, *www.graphviz.org*).

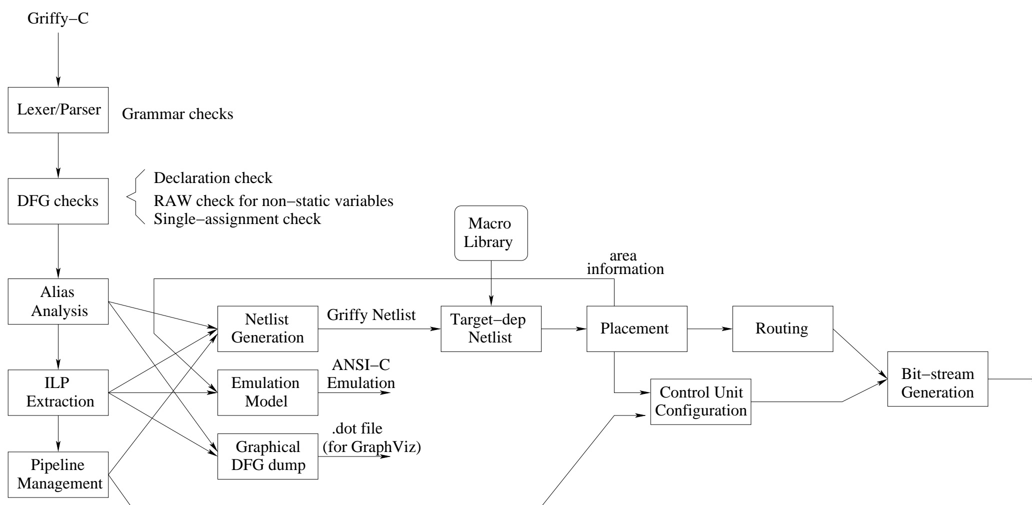


Figure 4.11: Simplified Griffy flow for PiCoGA-III

4.3 Target-specific customizations and back-end flows

Previous sections have described the Griffy flow for a generic target. The description has been focused on the exploitation of instruction level parallelism as a mean to organize DFG in pipeline stages. This is a common point for most of the reconfigurable architectures, although specific customization including checks or optimizations shall be implemented. In this section, it is provided a brief overview of two specific customizations. In the first case, Griffy-C is used to configure the PiCoGA (in particular, the description focus on the 3rd release, PiCoGA-III, included in the DREAM adaptive DSP), while the second target is a commercially available embedded FPGA for which a VHDL code is provided from a Griffy description in order to enter in the eFPGA proprietary tool-flow. Furthermore, for the PiCoGA is outlined the back-end tool-flow including the place & route and the bitstream kit.

4.3.1 DFG mapping for PiCoGA

PiCoGA-III is a reconfigurable device implementing pipelined data flow graph on a hybrid architecture in which computational parts are mapped on an island-style matrix of 16×24 4-bitwise tiles, while the pipeline management is handled by a dedicated pipeline control unit. For that, while the front-end, does not need a specific customization[3], but require a complete target-specific back-end flow. In fact, under some hypothesis on the connectivity, the netlist provided by the front-end is not target specific, and the specific PiCoGA customization only requires to add a physical mapping phase in which:

- each computational nodes is implemented using the resources available on one or more reconfigurable logic cells (RLCs), thus providing a target-specific mapping;
- each pipeline stage controller is implemented using the dedicated row control units.

Physical mapping is implemented using a library-based approach in which each Griffy computational node is split into one or more RLCs. No physical synthesis is performed, if we exclude the routing-only optimization implemented in the Griffy Front-End. In fact, Griffy code is intended as a structural way to effectively handle a reconfigurable device under a pipelined DFG paradigm, providing the programmer a low-level optimization step similar to the assembly-level optimization for DSPs. The target specific netlist is then placed on PiCoGA. Since each row can be fired synchronously, each row can be used at most by one pipeline stage, while more than one row can be triggered together in order to build larger pipeline stages. Placement is than the phase in which pipeline stages are split into one or more rows, and reconfigurable logic cells are fitted into. In order to contain the number of used rows, a pseudo-malloc algorithm is used: for each pipeline stage, computational nodes are sorted depending

[3]if we exclude a writeback alignment, that is implemented as an additional check condition during the candidate analysis phase.

on the relative weight (the number of RLC required for the implementation) and they are fit in the first empty space found in a free row or in a row assigned to the same pipeline stage. Reduction of connection costs is implemented applying simulated annealing internally to the pipeline stage, using a Manhattan metric as cost function.

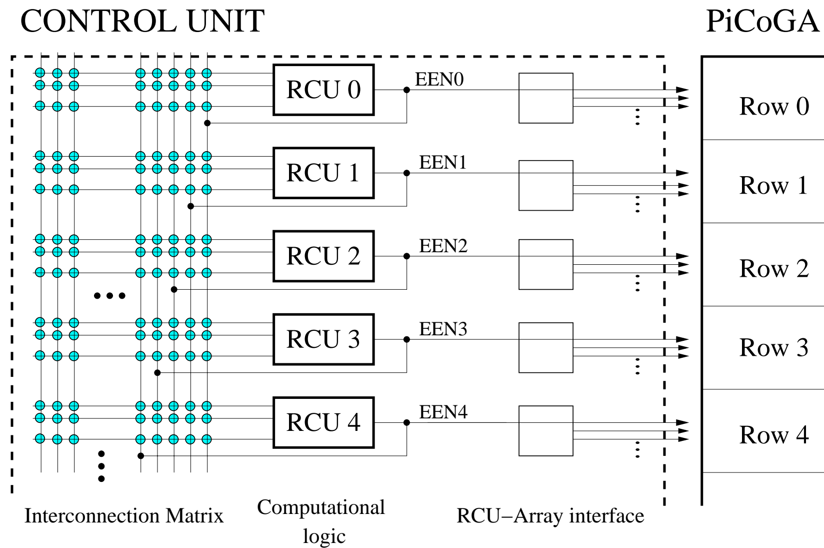


Figure 4.12: PiCoGA-III control unit programmable interconnect

After the placement phase, the pipeline control unit is configured. In particular, the configuration of every pipeline stage controller is applied to each row used to implement a pipeline stage. As shown in Fig. 4.12, a dedicate programmable bus is provided the necessary handshakes, propagating the execution enables between predecessor and successor nodes. When more than one row is used to implement a single pipeline stage, Griffy tools perform the connection with the nearest row for both preceding and successive pipeline stage, in order to reduce the routing utilization.

PiCoGA routing is programmed using a customized version of VPR [81], a well-known open-source tool developed at the University of Toronto. It is based on a state of the art timing-driven negotiation-based pathfinder algorithm in which resources over-utilization is allowed in the first iterations of the routing algorithm. The final solution is achieved minimize

the overall cost that is basically driven by the Elmore delay associated to the nets increased. To avoid over-utilization, an additional cost parameter is introduced in order to increase the cost of overused resources, that become less appealing and that shall be negotiated among the "users". On this context, PiCoGA specific customizations are focused on the architecture modelling, while the routing algorithm is not changed with respect to the basic one proposed in VPR. In particular, it was modelled the 2-bit granularity of the interconnections and the particular switch-block [66].

After place & route the area required for the specific Griffy operation is available and is reported to the simulation engine, in order to verify the resources utilization on PiCoGA. The routing part is necessary in order to take into account the routing exceeding the bounding box defined by the placement, as could happen in the case of design with high routing congestion.

The last step of the PiCoGA specific tool-chain is the generation of the bit-stream. This topic is achieved in two steps. In the first phase both RLC configuration (from the physical mapping) and routing configuration are translated to a textual bit-stream that specifies the logical value of each programmed bit. Only in a second phase bits are placed corresponding to the physical implementation of the device and the configuration bit-stream is generated in the form of a C vector.

4.3.2 DFG mapping for eFPGA

This section describes the mapping of Griffy-C code in device that have a proprietary back-end flow receiving, as entry-point, standard hardware description languages. In particular, this section describes a Griffy target generating VHDL code implemented, as a prototype, in order to provide the XiSystem architecture a homogeneous algorithm development environment.

The XiSystem architecture [67] is the first time architecture integrating two different field-programmable devices to provide application-specific computing blocks and IOs. A XiRisc reconfigurable processor is exploited

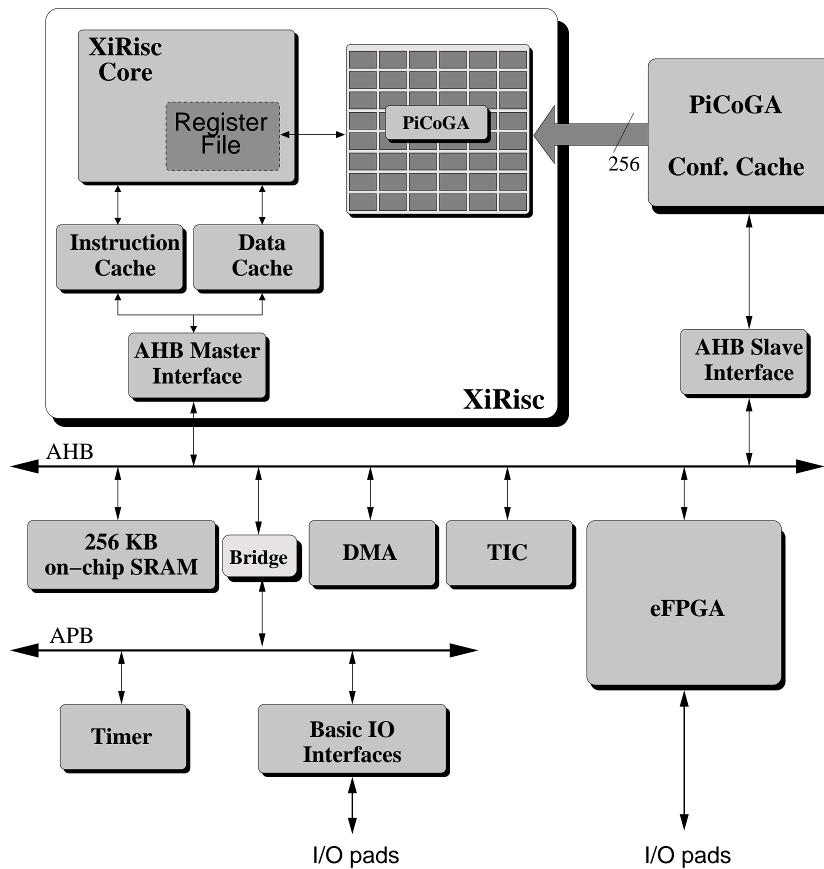


Figure 4.13: XiSystem SoC architecture

to achieve a more than one order of magnitude speed-up and energy consumption reduction vis-à-vis a DSP-like processor, while an embedded FPGA (eFPGA) is integrated in the system in order to make it flexible enough to support various IO ports and protocols. The reconfigurable IO device is also utilized for pre/post data processing and implementation of some standard computational blocks. Fig. 4.13 shows the overall system architecture.

While instruction set extensions for the XiRisc processor is generated configuring the PiCoGA starting from Griffy-C by the previously described flow, the management of the eFPGA could depend on the specific utilization. On one hand, hardware description languages provide the designer a straightforward way to describe I/O protocols directly exposing timing issues. On the other hand, if the eFPGA is used for pre/post data pro-

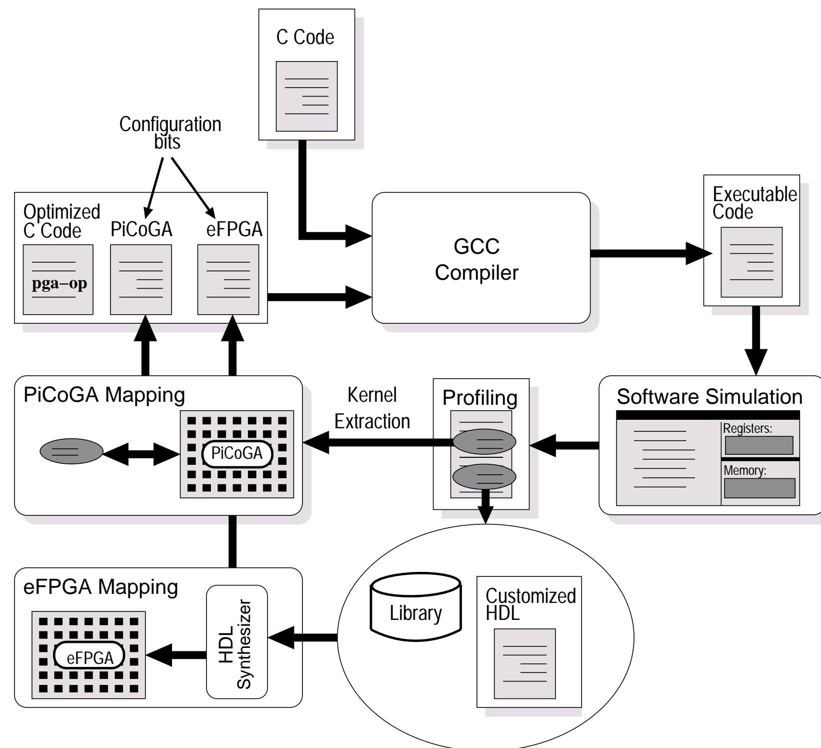


Figure 4.14: Overall software tool-chain

cessing or as a streaming computational block the utilization of hardware description languages could be substituted with the same high level description language (Griffy-C) used for PiCoGA configuration. Exploitation of both instruction and data parallelism are key elements to achieve impressive performance improvement also using standard reconfigurable devices, whereas the utilization of software-like languages could be intended as a way to allows software programmer to benefit from this technology. Also in this case, the detection of critical kernels is made through an iterative profiling step, where the programmer can evaluate various possible implementations of the algorithm. Moreover, the programmer can choose to implement such a kernel on PiCoGA or on eFPGA (or in both), as well as to choose what kind of approach (Griffy-C or VHDL) for the programming of the eFPGA. Fig. 4.14 shows the overall software toolchain implemented for the XiSystem architecture.

In particular, the translation of the Griffy-C code in VHDL is based on

	PiCoGA			eFPGA		
Kernel	Area mm^2	Contexts used	Eff. $\frac{MHz}{mm^2}$	Area mm^2	Freq. MHz	Eff. $\frac{MHz}{mm^2}$
Motion	11	4	15.09	10.76	52	4.81
Pred.	5.5	2	30.18	2.54	86	33.66
FDCT	7.3	2	22.64	3.18	68	21.26
Quant.	10.5	3	15.75	3.81	47	12.4
IDCT	7.3	4	22.64	9.14	46	5

Table 4.1: PiCoGA vs. eFPGA computational efficiency comparison

the same principles adopted for PiCoGA programming. A library based generation of appropriately sized computational blocks (i.e. adder, subtractor, bit-wise logic, etc.) and a library based generation of pipeline stage controllers. A library of VHDL components implementing both computational and control parts has been developed and the back-end flow instance the appropriate component thus realizing the target specific netlist. With respect to the mapping on PiCoGA, only one pipeline stage controller is generated since eFPGAs are not organized by rows. Moreover, the handshaking among pipeline stages is handled by point-to-point direct connection. The final design mix both computational and control parts in the same support, differently from PiCoGA approach in which computation and control are separated.

The realization of the prototype tool-flow generating VHDL code starting from Griffy-C description allowed a comparison between the PiCoGA and the eFPGA. The comparison has been conducted on the MPEG2 encoder application evaluating the maximum performance achievable on critical kernels. Table 4.1 shows the area occupation, working frequency and computational efficiency of the same 5 circuits mapped on both devices starting from the same Griffy-C code. In the case of the PiCoGA (ver 1.0, integrated in XiRisc) a fixed frequency of 166 MHz was considered. Computational efficiency has been calculated as the ratio between the working frequency and the area occupied on the device to implement

Circuit	eFPGA area occupation	Frequency
IEEE1284	1.5%	83MHz (SCK/2)
RS232	39%	83MHz (SCK/2)
I^2C	8%	55MHz (SCK/3)
LCD + YUV-RGB conv.	28%	42MHz (SCK/4)
VideoCam	1.5%	166MHz (SCK)
CRC	32%	55MHz (SCK/3)
Reed-Solomon	20%	55MHz (SCK/3)
IDCT	60%	42MHz (SCK/4)

SCK: System CK frequency

Table 4.2: Area occupation and working frequency of circuits mapped on the eFPGA

the circuit. The PiCoGA advantage is maximum when all the 4 contexts are used, as in the case of motion estimation and IDCT which are the most critical kernels of MPEG2 encoder. This demonstrates that PiCoGA can be 2 to 3 times more efficient than the eFPGA when implementing DSP algorithms, since the introduction of the eFPGA is not aimed at increasing computational density but at adding system interfacing flexibility and at providing additional parallel pre/post processing facilities.

Common communication protocols, such as I^2C, RS232 and IEEE1284, have been mapped on the reconfigurable IO module, satisfying the specific requirements of each protocol (see Table 4.2). Moreover, it could be convenient to map some additional post/pre-processing operations, for example to perform data format manipulation or error correcting codes, which are well suited to FPGA implementation because of their bit-level granularity. This allows one to remove a portion of computational load from the central processor. Particularly interesting are error detection and correction algorithms such as CRC and Reed-Solomon, which is almost ubiquitous, used for example in storage drives, wireless communications, digital television, and broadband modems. Both have been mapped in the

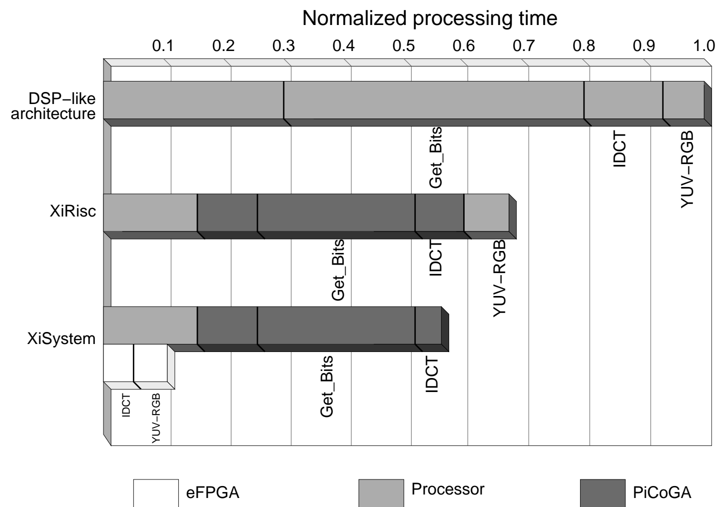


Figure 4.15: XiSystem MPEG2 decoder performance

eFPGA, which is capable of supporting up to a 100MB/sec data rate for the Reed-Solomon encoder.

As a proof of the benefit provided by mixed reconfigurable devices, it has been developed an MPEG-2 application (encoder and decoder) on XiSystem, applied to a standard QCIF stream with a frame resolution of 176x144 pixels and half-pel precision. Application-specific instructions introduced with the PiCoGA achieve a 5x speed-up and 66% energy reduction on the encoder and 1.5x speed-up on the decoder. Reconfigurable IOs are used to implement drivers for external peripherals such as an LCD display or a videocam, not binding the chip to any specific device. Since the LCD display chosen requires RGB pixel format while an MPEG decoder computes frames in YUV format, the necessary conversion has been implemented in the eFPGA, removing this procedure from the central core. This simple data post-processing achieved a 10% speed-up and 6% energy saving on the whole decoder application. Moreover the coprocessor configuration of the eFPGA was used to implement the row processing part of IDCT, achieving a further 6% speed-up, as shown in Figure 4.15.

Chapter 5

Simulation of dynamically reconfigurable processors

"Modern processors are incredibly complex marvels of engineering that are becoming increasingly hard to evaluate" [89].

In the case of reconfigurable architectures, simulation and performance evaluation shows additional complexity due to the coupling of dynamically programmable logic with standard processor cores. On one hand, fast simulation is a strict requirement for a design environment in which the iterative refinement of partitioning between hardware and software is a critical point in order to achieve the expected performance improvement. On the other hand, fast simulation requires the instruction set customization to be handled using high-level models (it is not required to control each transition associated to each signal, but only the overall behavior) and new strategies are thus necessary in order to save both cycle-level accuracy and fast reconfiguration time.

Design of reconfigurable architecture simulators can be oriented to fast source-level retargeting or to dynamic simulator extension. In the first case, the Instruction Set Simulator (ISS) description allows the user to rapidly add and remove instructions from the basic instruction set, but it requires the simulator to be recompiled for each instruction set extension. As an example, the Language for Instruction Set Architecture description (LISA [90]) was introduced in order to reduce retargeting time in design

and exploration of processor architectures, even if instruction set extension involves detailed description of new pipeline irregularity, such as the case of different pipeline depths and internal stalls. Also the SimpleScalar Toolset [89] allows the user to modify both the instruction set and the pipeline architecture. This infrastructure has been used in [91] where reconfiguration is performed via application specific re-targeting and simulator recompiling (Fig. 4 in [91]).

A different approach is followed in [92] where speed-up and performance estimation involve the GNU **gprof** profiler and a prototype FPGA used to synthesize application kernels. Due to space limitations, no file, I/O, or operating system calls have been implemented on the prototype FPGA. Kernels are implemented on the prototype FPGA allowing to compare execution time for the reconfigurable implementation to software implementation. The application speed-up is estimated employing the Amdahl's law.

In the Griffy project, two layer of simulation environment has been provided:

- *functional simulation*, in which the programmer can validate a Griffy-C code on the host-machine (e.g. x86 Linux) using a compiled simulator, which joins the Griffy emulation and processor code in the same executable.
- *cycle-accurate simulation*, in which the programmer can simulate the Griffy code on the target architecture, thus allowing both target specific debugging and performance evaluation.

In both the cases, the emulation of Griffy code, be it used for PiCoGA configuration or for different devices like an eFPGA, is automatically generated by the Griffy toolchain. The following of the chapter explains the Griffy emulation principles, with an example of integration in an open-source environment. As a test-case will be considered the GNU GDB-based simulation environment of the XiRisc processor. GNU GDB is not a cycle accurate simulation environment, like that ones provided for example by LISA, but it is a commonly available and well-know tool. For this

reason, it will be used as an example of integration, although the Griffy emulation has been successfully plugged also in LISA-based environment for both XiRisc and DREAM processors.

5.1 Functional simulation

The goal of the functional simulation is to provide the end-user a tool for the verification and the debugging of the Griffy-C specification. For this reason one of the most important points is the speed of simulation, since the verification could require intensive tests with a high volume of data. In this case and for this purpose, it is not so important to have a correct management of all the timing aspects (for example, the pipeline evolution), which on the contrary will be required in target specific performance evaluations. Two aspects will be considered in following:

- the functional emulation of Griffy-C code, which requires the translation to standard ANSI C of the Griffy-C operations, including both the operators and the #pragma attributes.
- the definition of a virtual platform (a simple reconfigurable architecture) which allows the utilization of functionalities defined by Griffy-C code in a standard C code.

5.1.1 Functional emulation

Functional emulation is the standard C translation of the Griffy-C operations, including both operators and #pragma attributes. Griffy tools provide an emulation function for each Griffy operation defined in the source code. Emulation is based on a re-ordered Griffy-C code scheduled per pipeline stages in order to validate also the consistency of the data dependencies.

The translation of Griffy C code is handled in two steps:

- *mapping of Griffy-C operators on standard C operators.* While for many operators this phase is a pure cut-&-paste, since most Griffy-C op-

erators are a subset of ANSI C operators, additional functionalities requires the generation of specific code. As an example, LUT emulation is obtained declaring local arrays to implement the functionality, while the operator is substituted with the read of an array elements.

```
#pragma attrib out SIZE=1;
#pragma attrib in  SIZE=4;
out = in @ 0x05;

      comes

{
  unsigned char emulation_vector[16] = { 1,0,1,0,
                                         0,0,0,0,
                                         0,0,0,0,
                                         0,0,0,0};
  out = emulation_vector[in];
}
```

- *adding attribute side effects.* Attributes allow to both define bit-level variable size and extract information about the carryout and overflow condition on arithmetic operations. They are handle adding a "normalization" process after the basic operation in order to *patch* the result. N-bit resizing is implemented by means of masking and depending on the variable type is obtained by:
  - bit-wise and with $2^N - 1$ in the case of *unsigned* variables.
  - left and right shift by $M - N$ in the case of *signed* variables, where M is the original size (32-bit for integers, 16-bit for short and 8-bit for char types).

  Carryout and overflow informations are extracted adding specific emulation code after the operation under check. As an example, for an addition:

  - *carryout* is obtained by:

```
tmp1 = i1 + i2;
__griffy_gen_carryout_tmp1 = (( ( (i1 >> 1) & 0x7fffffffULL ) +
               ( (i2 >> 1) & 0x7fffffffULL ) +
```

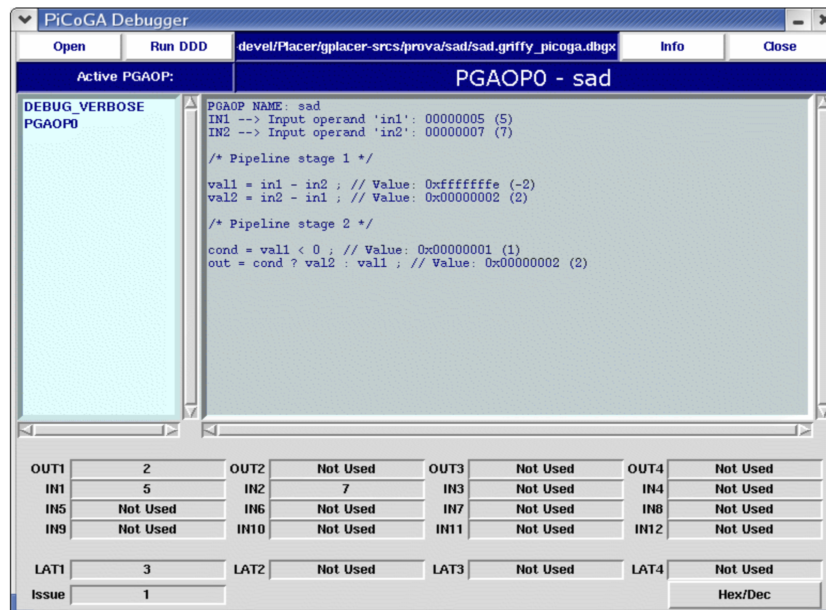


Figure 5.1: Griffy code viewer

```
( (((i1 & 0x01) + (i2 & 0x01)) >> 1) & 0x1 )
) >> (32 - 1)) & 0x01;
```

– *overflow* is obtained by:

```
  tmp1 = i1 + i2 ;
  __griffy_gen_overflow_tmp1 = (
( ((i1 < 0) ? 1 : 0) == ((i2 < 0) ? 1 : 0) ) &&
        ( ((i1 < 0) ? 1 : 0) != ((tmp1 < 0) ? 1 : 0) ) ) & 0x01;
```

After each Griffy-C emulation, a file dump is performed in order to allows the inspection on the internal value. Since Griffy code describes a hardware part (although reconfigurable), it could be misleading to allow break-pointing on Griffy code just like on standard C code. For this reason, intermediate results are only reported in a dedicated viewer (Fig. 5.1) available for the entire simulation environment in which Griffy emulation is plugged.

5.1.2 Reconfigurable devices management via virtual target

The verification of Griffy-C code in a given application requires to perform several operations on the reconfigurable device programmed by Griffy-C. Most of them are strongly connected to the specific system including the device, the way in which the device is connected to the processor and the specific management protocol of the device. Usually, from a hardware point of view, a reconfigurable device is explicitly triggered to load configurations and to execute operations, writing the specific commands in a specific hardware port. While the physical implementation of these operations is roughly system independent, the mechanism that generate the command is strongly dependent from the system around the reconfigurable engine. In this context, we can suppose, without loss of generality, that the reconfigurable device is managed through a standard microprocessor. For these reasons, we provide a virtual Application Program Interface (API) that allows the user to load, trigger, and deallocate operations on the reconfigurable device in a sort of virtual platform. In the following, this section describes the functional simulation model used to handle Griffy operations through a standard processor, as in the following example.

```
Initialization { ... }
PD = pga_allocate (my_first_pgaop);
{ ... }
Computation { ... }
For( ; ; ) {
  ...
  pgaop_direct1(PD, &outputs,... ,inputs, ...);
  ...
}
Conclusion { ... }
pga_dealloacate (PD);
```

Low level built-in functions are provided to manage the configuration. `pga_load` and `pga_free` builtin functions allow users to load a configuration and to release the space used by a configuration. They are low level primitives to handle a reconfigurable device like the PiCoGA that provides the user the possibility to load more than one configuration in the same device. Although it is provided an emulation of these builtins, their functionality needs to be considered, for most devices, as atomic. Using pga_load and pga_free, the user is responsible to allocate Griffy operations into the reconfigurable and to check the space availability, since this parameter is specific to the target. But, on the other hand, if the allocation is handled by a processor, it is also possible to run a flexible allocation mechanism, requiring the user to specify only the name of the Griffy operation (or pga-op, programmable gate array operation) to be load or free. This function can automatically find an appropriate empty space (layer and starting row) inside the reconfigurable device, thus providing the user an abstraction layer with respect to the direct hardware level management.

At the highest abstraction layer, the allocation mechanism is very similar of that one used for the dynamic memory allocation. The pga_allocate provides the user the capability of load a new configuration on the reconfigurable device. The pga_allocate function searches for an empty available space in the array, starting from the first row in the first configuration context (or layer), but it does not perform any analysis about fragmentation of the reconfigurable device. If the pga_allocate finds a proper space, it forces the pga_load command, which is the physical operation that really interact with the reconfigurable device. Of course, the user can directly force an allocation manually, using the pga_load, but in this case the allocation structure used by pga_allocate is not updated. The user is responsible to assure the data structure consistency when a mix of pga_load and pga_allocate is used. pga_allocate returns a pga-op descriptor (PD in the example before), that includes information about the location inside the reconfigurable device. The pgaop is triggered specifying the PD in the pgaop_direct1.

Of course, it is necessary to require the configuration of a given operation before its issue. Usually though, the configuration of an operation can be performed in the initialization phase of the application, and this rarely causes any overhead on the performance. The pgaop_direct1 is the link that allows to compute the previously described emulation function of a given PGAOP. It works like an ANSI C procedure, thus results are provided through memory referencing (pointer). The pgaop_direct1 requires:

- PD : PGAOP DESCRIPTOR
- 4 outputs (even if not all used)
- 12 inputs (even if not all used)

Communication with the reconfigurable device is handled by a virtual mechanism implemented in software by a function call. Real data transfer shall be defined and refined on the target-specific simulation environment. The pga_deallocate function allows to remove the specified operation (specified through the PD) when it has no longer to be used, or when the user needs space for a new set of operations. The pga_deallocate updates the same data structure used by pga_allocate (it's its inverse function), and perform a call to the pga_free. Before an allocation, the identification of a given operation is obtained referring to the name used in the Griffy-C declaration. When one or more operations are compiled by Griffy Tools, an enumerate list is provided, assuring the consistency of data structures (e.g. internally to pga_allocate). In the case of operation holding an internal state (by the utilization of static variables), the (re)initialization can be forced by the pga_init function. At the next execution, initialization values are reloaded in each static variable.

5.2 Instruction set extension through dynamic libraries

The basic idea of the Griffy simulation environment is to work as a plug-in for standard simulation environments, be its functional or cycle-accurate.

To maintain a high simulation speed, operations implemented on a reconfigurable device are emulated (as explained before), do not performing simulation at bit-level. Starting from emulation functions, the reconfigurable device resources are modelled in such a way that all the physical constraints are respected all along the program execution. In the case of cycle-accurate simulation, timing issues, as for example the pipeline evolution, are handled by a third additional wrapper.

Dynamic reconfiguration is thus handled by changing the emulation function associated to a specific pgaop descriptor (this link is provided by the execution of pga_load/pga_free primitives). Since for reconfigurable processor most of the architecture is defined and only a part changes depending on the application specific customization, reconfigurability is provided by means of dynamically linked libraries. This avoid the need of instruction set simulator (ISS) recompilation, that could be excessively time expensive for the end-user.

Therefore, the emulation of a set of functionalities mapped on the reconfigurable unit implies a description of:

- the functionalities to be implemented on the array;
- the description of the resources available on the reconfigurable device;
- the way the operations are structured within pipeline stages.

Such items are partly described in the ISS definition, and partly come with the dynamic library produced by the application compilation. Resources description is specific for the reconfigurable device, as well as the way in which pipeline evolution is handled, although the pipeline control structure can be considered common to all the operations. Therefore, it is possible to integrate these structures in the ISS core, by programming the pipeline manager for each new functionality loaded in the reconfigurable device. The dynamic library only needs to describe the pgaops functionality and the proper pipeline activation events. According to this approach,

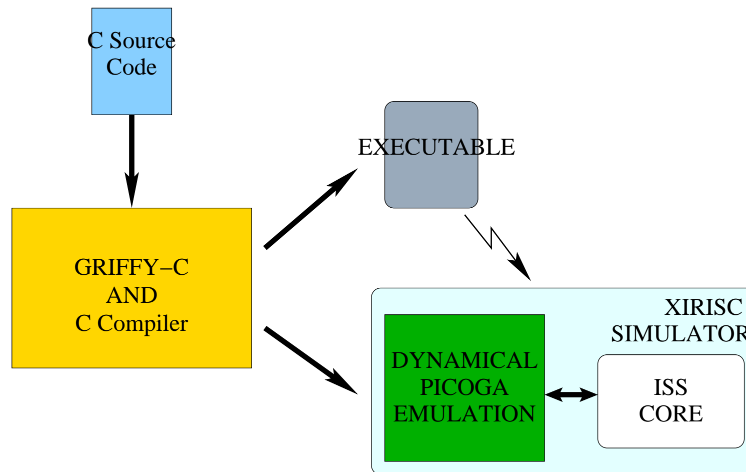


Figure 5.2: Simplified XiRisc simulation structure

as summarized in Figure 5.2 for the XiRisc processor, when a new application is compiled, the Griffy toolchain provides:

- an *executable program*, in ELF[1] format, composed by the processor code/data and the configuration bitstream for the reconfigurable device;
- an application-specific *emulation library* for the instruction set extension that is dynamically linked to the ISS-core.

The simulation library can be compiled in *verbose* mode. In this case, it is possible to monitor the internal status of the reconfigurable device during the ISS elaboration and to visualize it with an external viewer in order to implement source-line debugging.

As an example, we can consider the dynamic instruction set extension of a GNU GDB-based simulator. Similarly to other instruction set simulators, the core of GDB is the instruction set description in which a set of functions associates a specific functionality to each assembly operation. In the case of GDB, the `.igen` format is used in order to specify the instruction template (opcode + input and output registers), the mnemonic

[1]Executable and Linking Format

(dumped if simulation tracking is enabled) and the functionality. The following code shows the description of a simple ADD operation.

```
000000,5.RS,5.RT,5.RD,00000,000000:SPECIAL:32::ADD
"add r<RD>, r<RS>, r<RT>"
{
    GPR[RD] = GPR[RS] + GPR[RT];
}
```

RD represents the destination register, whereas RS and RT are the input registers. GPR is the general purpose register file. The first two lines represent respectively the instruction template and the mnemonic. The functionality, written in ANSI C, can include additional operations, such as for example the updates of the program counter in the case of conditional branches, delay slot management and so on. In the case of instruction set extension in which the functionality is defined by the end-user and often depend on the application, the retargeting performed re-compiling all the simulation engine is not particularly appealing. In the Griffy project, and in particular in the case of the XiRisc reconfigurable processor, dynamic instruction set extension is handled using dynamically linked libraries. In this case, the `.igen` description shall be modified to call an emulation function. In this case, during the compilation of GDB, a *curses*-library is used, providing error message for the invocation of undefined functionalities. When the application and the relative set of Griffy operation are defined, the emulation library providing correct emulation is available. GDB is then modified in order to support a-specific assembly instruction, instruction skeleton in which the functionality is specified at execution time, like in the following PGA32 code:

```
111110,5.RD1,5.RD2,5.RS,5.RT,4.PD,00:SPECIAL:32::PGA32
"pga32 <PD>, r<RD1>, r<RD2>, r<RS>, r<RT>"
{
  // Verification of PD availability
  if ( PGA_ID_table[PD] != 1 ) {
    fprintf (stderr, "Error!!! PD 0x%x not loaded\n", PD);
    exit (1);
  }

  // Dynamically linked with libemu.so
  __pga_emul[PD] (&latency_dest, &issue_delay,
```

```
    PD,
    &(GPR[RD1]), &(GPR[RD2]),    // Output list
    GPR[RS], GPR[RT]             // Input  list
   );
}
```

The __pga_emul is a vector of pointer to function that is initialized by the pga_load. During the pga_load, the emulation function is associated to the PD, and at the trigger is verified that this link exists. Input and output values are passed depending on the model of computation used: in this case, a functional unit model is chosen and data are passed through the register file (2 inputs and 2 outputs). Latency and issue delay represent the static parameter of the function and they are used in the timing model as discussed in the following.

5.2.1 Cycle-accurate simulation model

The management of a set of custom operations mapped on user-defined pipelines over a reconfigurable device is described in the following proposing at the beginning a model for a single configurable pipeline and than the same model will be extended to handle stalls involved, for example, in context switches, writeback conflicts and so on. As introduced in previous chapter, the computation of custom pipelines generated by Griffy tools is controlled through timed Petri Nets, with *operations firing* associated to each taken transition. In simulation, the check of the token availability for every node would require a large amount of time. To overcome the problem, it is proposed a different cycle-accurate model based on resource allocation vectors.

We distinguish between two levels of description of the custom pipeline:

- the “functional model”, in which we describe the functionality of the DFG, its area occupation and load penalty on the reconfigurable device.
- the “timing model”, which takes into account stalls occurrences both

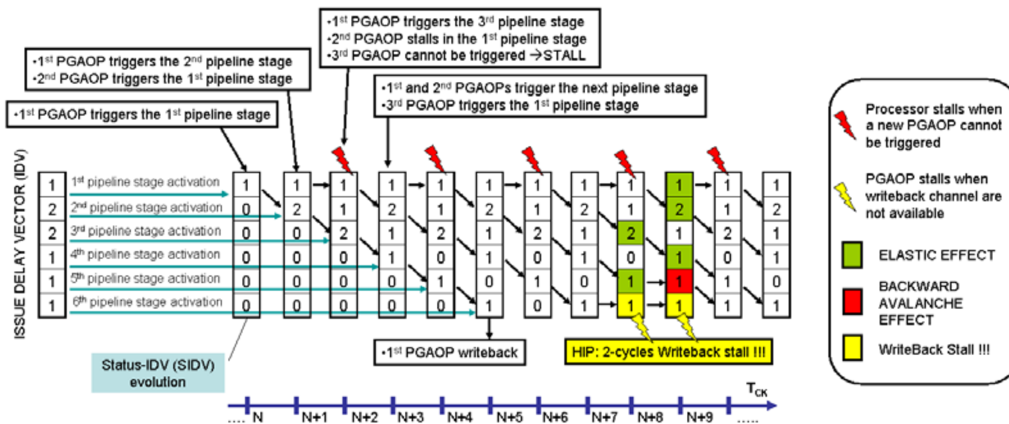


Figure 5.3: An example of pipeline evolution

inside and outside the reconfigurable device and due to data dependencies in the program flow and/or between successive `pgaop`s.

Although for debugging purposes it is possible to attach the functional model to a functional-only simulation engine (e.g. the *Gnu Debugger GDB*), a cycle accurate performance evaluation requires to interfacing the processor core ISS with the timing model in order to represent correctly all stalls. As an example, stalls in the XiRisc computations can be due to two different factors: inter-operation data hazards, due to dependencies in the processor program flow, and intra-operation pipeline hazards inside the PiCoGA computation.

Program Flow Hazards

A cycle accurate model can be seen as an abstract object with an internal hierarchy:

- the functional model is an internal, compilation dependent and dynamically linked *core* that describes the `pgaop` functionality;
- the timing model is a fixed *wrapper* handling communication with the ISS core.

The *wrapper* emulates pipeline activity, and handles all hazards occurrences thus allowing cycle-accurate simulation. It is also the *wrapper* that

appropriately calls the functional model of the required `pgaop`, writing back on the register file computation results at the appropriate time and issuing stalls according to the data hazard handling rules.

PiCoGA Internal Hazards

A relevant feature of the PiCoGA unit is the capability to compute concurrently on a pipelined pattern multiple issues of the same pgaop, resolving dynamically at computation time all potential hazards. Thus, a fundamental issue of the timing model embedded in the *wrapper* is the description of the pipeline management *inside the PiCoGA* in case of multiple issue of a given `pgaop`. Other architectures does not provide this feature, as well as they could not have the capability to load more than one operation at time. For all these architecture, the simulation model shall be simplified in term of concurrency management, whereas for example checks on resources availability shall be improved.

For each pipeline stage, preceding and successive data dependencies define an issue delay which describes the minimum number of cycles among successive firing. These "Pipeline effects" are described by an Issue Delay Vector (IDV), produced by the DFG compilation. The IDV describes for each pipeline stage (i.e. set of concurrent DFG nodes) the number of cycles, in the overall pipeline flow, during which the stage must be idle, because:

- it is waiting for an input not yet produced by a previous stage;
- one output must be processed by a following stage whose computation that has not yet been triggered.

The maximum value in the IDV will describe the issue delay of the overall pipeline that is the rate at which new issues of the same operation can be fed to the pipeline. An optimal DFG has an IDV composed of '1' for each stage. This means it features an issue delay of 1, and a new operation can be fed to the pipeline at each cycle (provided it does not cause program flow inconsistencies at processor level as described in section 5.2.1).

Of course, the IDV is specific for a given `pgaop` and represents the key information for the timing model.

During simulation, to each loaded configuration corresponds a Status-IDV (SIDV). The SIDV is used to verify resource availability: when a `pgaop` computes the ith pipeline stage, it sets the ith entry of the SIDV to the corresponding Issue Delay. At every cycle each element in the SIDV value is decremented: each specific `pgaop` issue will be stalled in the stage until the corresponding SIDV entry returns to zero. Figure 5.3 shows an example of pipeline evolution under this model. Let's suppose that the processor core is capable of trigger a new `pgaop` at every clock cycle. At $T_{ck} = N + 2$ we try to trigger a new `pgaop`, but a stall occurs because the pipeline stage 2 features an issue delay of 2.

The distance between successive operations in the pipeline ***is not fixed***: a stall condition occurs when a `pgaop` tries to compute a pipeline stage (i.e. DFG node) corresponding to a non-zero value in the SIDV. As a consequence, successive issues will (i) stall if the pipeline is at the maximum issue rate (backward avalanche effect) or (ii) proceed until they "reach" the stall location (elastic effect).

All effects discussed so far are due to multiple issues of the identical `pgaop`: this guarantees that at most only one issue per clock is computing the last stage of the pipeline and is thus performing writeback on the register file. On the contrary, concurrent computation of different `pgaop`s, that is `pgaop`s implementing *different* pipelines on the PiCoGA resources, are completely independent and feature different latencies. Consequently, they may cause conflicts on the writeback channels. This will cause a stall of one of the two pipelines.

Another cause of stall handled by the *wrapper* is due to context switch. For architecture like PiCoGA featuring multi-context capabilities, the reconfigurable device is able to change the active configuration contexts in few cycles. In the case of PiCoGA, the active context (among the 4 available) can be switched in a single clock cycle simply addressing a `pgaop` that is residing in a different context with respect to the active one. Several stalls may then be necessary in order to complete all current computa-

tions in the current context and flush all active pipelines before the context switch.

Of course, all internal stalls described so far may affect the processor core pipeline, as the reconfigurable device may refuse the issue of an operation whose initial stage is already occupied by a stall. Furthermore, an incorrect utilization of the reconfigurable device may cause exceptions. The wrapper is capable to back-annotate stall (and exception) information to the processor model.

In Figure 5.3, a couple of stalls occur during the execution when the reconfigurable device tries to writeback values to the register file ($T_{ck} = N + 8$). In the following clock cycle ($T_{ck} = N + 9$) an *elastic effect* occurs, allowing computing other pending operations until they reach the stalled pipeline stage: successive issues will "crowd" in the stages following the stall. A backward avalanche stall will be caused. When the first stalled stage will resume computation, all following pipelines will in turn resume their normal computation and spread over the pipeline. In general, both backward avalanche and elastic effect can occur when more than one operation is under execution in reconfigurable devices like the PiCoGA, featuring a flexible pipeline management based on the Petri-Net paradigm. The resulting overall effect resembles the alternance of compression and dilation phases in longitudinal wave propagation.

5.2.2 Simulation speed analysis

Evaluation of a simulation engine needs to take into account several parameters such as flexibility and accuracy. As well, for a reconfigurable architecture we need to know how the ISS can be retargeted and how much time must be spent in retargeting. As additional constraint, the design exploration needs to have a fast simulation engine because of the huge amount of time spent from the end-user during the application development. In the case of the simulation engine described before, reconfiguration costs are tightly coupled with the time spent compiling the application. In fact, when the hardware-part of an application is compiled, the

Algorithm		LISA-System	LISA-core	GDB
FDCT	#CK Cycles	11572872	8225712	7028208
	Sim. Time	160 sec.	7 sec.	7sec.
IDCT	#CK Cycles	18492536	15279930	14237048
	Sim. Time	300 sec.	14 sec.	13 sec.
Quantization	#CK Cycles	33727336	25929123	21899910
	Sim. Time	528 sec.	24 sec.	21 sec.
VLC	#CK Cycles	34663765	34132828	28611213
	Sim. Time	532 sec.	33 sec.	27 sec.
Motion Estimation	#CK Cycles	815167602	805077673	695172321
	Sim. Time	210 min.	754 sec.	650 sec.
MPEG-2 Encoder	#CK Cycles	978423450	920866413	795349022
	Sim. Time	260 min.	845 sec.	747 sec.
IDEA	#CK Cycles	84682673	84682662	76556209
	Sim. Time	22.4 min.	79 sec.	69 sec.
CRC	#CK Cycles	11779	11295	10245
	Sim. Time	1.15 sec.	0.5 sec.	0.5 sec.

Table 5.1: Simulation results (without PiCoGA)

compilation flow automatically provides the simulation library which represents the functional model of the current application and the amount of time required can be estimated in few minutes.

In order to trace results, it has been evaluated the performance of three simulation engines, the first one based on GNU GDB and two based on LISA ISS, all featuring dynamic instruction set extension through the Griffy-generated emulation library. The GDB model takes into account both reconfigurable unit latency and maximum issue delay of each pgaop, but does not consider processor internal stalls. The LISA-core adds an accurate evaluation of processor stalls due to internal pipeline and the reconfigurable unit timing model described in the previous section. The LISA-System simulator is the more accurate engine that integrates LISA-core modelling both contentions on bus architecture and latency of memory hierarchy described in System-C.

Algorithm		LISA-System	LISA-core	GDB
FDCT	#CK Cycles	6354872	4446330	3936240
	Sim. Time	94 sec.	5 sec.	4 sec.
IDCT	#CK Cycles	13097695	13097695	10207483
	Sim. Time	186 sec.	13 sec.	10 sec.
Quantization	#CK Cycles	21437286	12662704	10929703
	Sim. Time	324 sec.	13 sec.	10 sec.
VLC	#CK Cycles	24303819	24194040	21686592
	Sim. Time	340 sec.	25 sec.	22 sec.
Motion Estimation	#CK Cycles	127151165	120938635	100900354
	Sim. Time	46 min.	123 sec.	104 sec.
MPEG-2 Encoder	#CK Cycles	239064880	192245109	163166038
	Sim. Time	90 min.	270 sec.	168 sec.
IDEA	#CK Cycles	31947675	31947688	28998568
	Sim. Time	490 sec.	34 sec.	26 sec.
CRC	#CK Cycles	8204	8198	5637
	Sim. Time	1 sec.	0.4 sec.	0.3 sec.

Table 5.2: Simulation results (with PiCoGA)

Referring to the XiRisc model, Tables 5.2 and 5.1 show performance results achieved from the ISS running on a Sun Sparc UltraIII workstation, 900MHz with and without the reconfigurable unit (PiCoGA in the case of XiRisc) emulation engine compared to the functional-only GDB-based simulation engine running on a Linux workstation with Athlon XP2000+. The LISA ISS environment shows a computational capability of about 1 MIPS, while the integrated system model computes about 50,000 clock cycles per second, reducing the overall simulation engine performance up to 20 times. The simulation environment has been mainly benchmarked using a 12 QCIF Frames (176x144) MPEG-2 encoding requiring about 1 billion clock cycles to be executed. As a reference, a HDL simulation engine runs about 1000 clock/sec without a system-level integration and about 400 clock/sec with memory hierarchy. The LISA-System cycle-accurate instruction set simulator runs about three orders of magnitude faster than

HDL simulation, while the LISA-core gains up to five orders of magnitude with respect to HDL simulation time.

Chapter 6

Application development on reconfigurable processors

I'm not a bit-level programmer,
but a right-level programmer!
(Mario Toma, STMicroelectronics)

Reconfigurable hardware accelerators are the strong point of reconfigurable processors if compared to general purpose processors (GPPs). Efficient programming of reconfigurable architectures often implies finding the best HW/SW partitioning of the C program execution between the standard hardwired functional units (SW) and the hardware accelerators (HW). Unlike C-to-FPGA synthesis which translates a *whole* C program to hardware and therefore needs to fully support all C constructs (arithmetic and logical operations, memory access, etc. [43, 38]), the compilation flow for a reconfigurable processor translates to programmable logic *only* the program sections that may benefit most and can be implemented as hardware, while the rest of the algorithm is executed on the hardwired functional units. Hence restrictions on C constructs that can be mapped onto HW do not compromise the overall system capabilities. Finding the optimal HW/SW partitioning for a given reconfigurable architecture is a very complex task for a software tool, and hence the Griffy development environment currently provides support only for manual partitioning. As

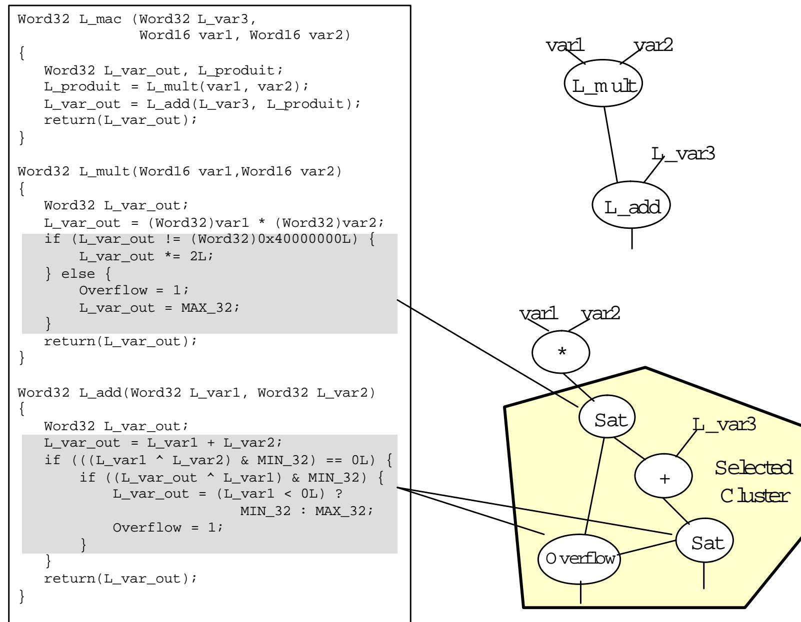


Figure 6.1: Case study: saturating MAC for low bit-rate audio compression

a general rule, control or data management is better suited to the sequential elaboration of the CPU core, while computational kernels with higher instruction level parallelism (ILP) or prevalent bit-level operations benefit most from a hardware implementation.

Let us consider, as an example, the case of saturating multiply-and-accumulate (MAC) used in many low bit-rate audio compression applications implemented on the XiRisc reconfigurable processor. A typical code is shown in Fig. 6.1. Since the PiCoGA is not well suited to implementing large multipliers, the initial multiplication is performed on the processor, while the other operations (the saturation of the multiplication and the saturating sum) are “clustered” to a single operation performed on the PiCoGA. The selected code is backgrounded in grey in Fig. 6.1 and a simplified data-flow graph is shown.

Fig. 6.2 shows the Griffy-C code relating to implementation of the selected kernel and the resulting mixed HW/SW saturating multiplication. The Griffy-C code is included between “pragma” directives and additional pragma directives are used to manage the size of each variable at the bit

```
#pragma picoga kernel_L_add_mux 2 3
               L_var_out overflow         // Output list
               L_var1 L_var2 overflow1    // Input  list
{
   int L_sum;
#pragma attrib L_sum SAT
   int Lvar1, Lvar1_tmp, Lvar2,Lvar2_tmp;
   int Lvar2_out_tmp, L_var_out_tmp1;
   unsigned char cond, cond1, overflow_a;
#pragma attrib cond, cond1, overflow_a SIZE=1

   Lvar2 = L_var2;
   Lvar2_tmp = Lvar2 << 1;
   cond = Lvar2 != 0x40000000;
   overflow_a = cond ? overflow1 : 1;
   Lvar2_out_tmp = cond ? Lvar2_tmp:0x7fffffff;
   Lvar1 = L_var1;
   L_sum = Lvar1 + Lvar2_out_tmp;

   cond1 = Lvar1 < 0;
   L_var_out_tmp1 = cond1 ? 0x80000000:0x7fffffff;
   L_var_out = L_sum(overflow) ? L_var_out_tmp1:L_sum;
   overflow  = L_sum(overflow) ? 1:overflow_a;
}
#pragma end

Word32
L_mac(Word32 L_var3, Word16 var1, Word16 var2)
{
   Word32 L_var_out;
   L_var_out = (Word32)var1 * (Word32)var2;
   kernel_L_add_mux ( L_var_out, Overflow,
                      L_var3, L_var_out, Overflow );
   return(L_var_out);
}
```

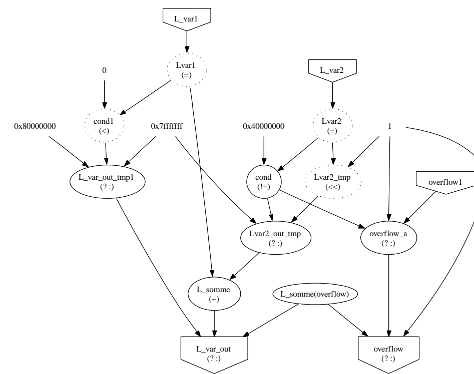


Figure 6.2: Case study: Griffy-C code for saturating arithmetic

level. In addition, the SAT flag enables a bit of overflow information to be extracted directly from an arithmetic operation. Control statements are dismantled in conditional assignments mapped one-to-one to multiplexers with 2 input words. The corresponding pipelined DFG is also represented: nodes are aligned per pipeline stage, thus showing the 4 resulting stages. On the processor side, the PiCoGA operation is triggered as a function-like call.

Fig. 6.3 summarizes the optimization process. While (a) is the starting point, and (b) represents the partitioning step, (c) corresponds to the final optimization. Analysis of the basic implementation in (b) shows that memory accesses to provide data, multiplication and the PiCoGA operation are cascaded in such a way as not to allow exploiting either the pipelining inside the array or the concurrent elaboration between PiCoGA and processor. However by applying a loop transformation based on standard software pipelining methodology [61, 59], it is possible to compose a loop body (shown in (c)) where the PiCoGA operation and the processor code run concurrently, since they are working on data referring to successive loop iterations. This is a first optimization step, which exploits the

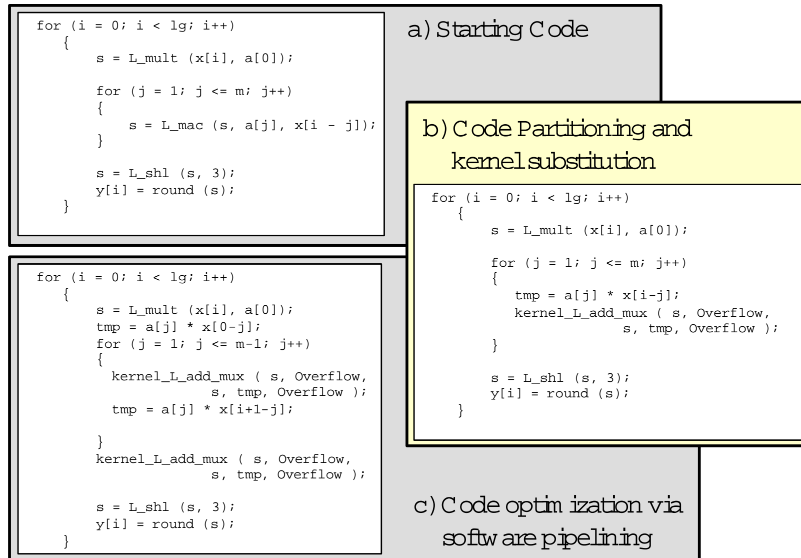


Figure 6.3: Case study: software pipelining across processor and PiCoGA

concurrency between the processor core and the PiCoGA, but further improvements can be achieved by applying loop unrolling or similar methods capable of enhancing the PiCoGA hardware pipelining as well [60].

For a wide spectrum of current embedded applications, the typical kernels are located in the core of innermost loops [57], which can be usually described with traditional data-flow graphs. Very significant speed-ups can be achieved by pipelining successive loop iterations in the case of acyclic graphs [59]. This software compilation technique [61] has been shown to be an effective means to increase the instruction level parallelism without increasing the code size for highly parallel processor architectures (e.g., super-scalar processors with 8 or more data channels). Reconfigurable processor architecture can also directly exploit more traditional hardware pipelining [60]. In fact, the innermost loop can be mapped (or *clustered*), for example, in a single Griffy operation and then executed overlapping successive iteration depending on the issue delay. Nevertheless, several application do not allow the clustering of the whole innermost loop because either the whole loop does not fit in the reconfigurable device or the whole loop features instructions ill-suited to the reconfigurable device (as the multiplication in the previous example). Furthermore, for

many architecture, the access to the memory is performed only by the processor core (reducing the cache coherency problems), that is thus responsible to feed data to the reconfigurable device and store data after the elaboration. Let's consider the following example:

```
// C source code

for (i=0; i<64; i++) {
  tmp1   = f1(v[i]);
  tmp2   = tmp1 * w[i];
  out[i] = f2 (tmp2,z[i]);
}
```

```
# Pseudo-ASM code
loop:
  reg3   = load (v[i]);
  reg4   = load (w[i]);
  reg5   = load (z[i]);
  tmp1   = f1   (reg3);
  tmp2   = mult (tmp1, reg4);
  reg6   = f2   (tmp2, reg5);
  out[i] = store (reg6);
loop i;
```

where `f1` and `f2` are computations suitable for the reconfigurable accelerator, while the multiplication is an example of an operation that is usually better implemented in a hardwired functional unit. Load/store operations are commonly implemented in the processor core, as in the case of XiRisc. This is an example of an acyclic graph in which a mixture of software pipelining and clustering can improve the performance of the application with respect to software pipelining or clustering taken alone. Superscalar processors apply software pipelining to increase instruction level parallelism up to the limit determined by the available processor resources. Traditional reconfigurable processors apply the clustering of assembly instructions moving the computation to the HW and waiting for the results, since both the communication overheads often forbid to spread the computation over a mix of processor and configware resources, and configuration tools often discard kernels with strange operators. In the "reconfigurable" computation pattern instead, we can map `f1` and `f2` as concurrent sub-graphs of the same Griffy operation and then apply a software pipelining schedule to the resulting code:

```
    # Pseudo-ASM code
    # SW-pipelining prologue
loop:
    reg3      = load (v[i]);
    reg4      = load (w[i]);
    reg5      = load (z[i-1]);
```

```
  tmp2      = mult (tmp1, reg4); # tmp1 is the old value
  out[i-2]  = store (reg6);
  # PiCoGA-operation implements both f1 and f2
  tmp1,reg6 = PiCoGA-operation (reg3,tmp2,reg5);
loop i;
  # SW-pipelining epilogue
```

The execution of the Griffy-operation can be concurrent to the processor computation and it can overlap both loop instructions and memory access. Furthermore, sub-graphs `f1` and `f2` work with data referred to different loop iterations in the original code, thus the concurrent execution of `f1`, `f2` and the multiplication can be achieved. As shown in this example, applicable for example to the XiRisc architecture, it is possible to design *"around"* the processor functional units an additional functional unit implemented as a Griffy operation, which is highly specific for the kernel.

For run-time reconfigurable devices, like the PiCoGA, it is possible to provide dedicated functional units for as many kernels as they can be found in an application, thus truly realizing the dynamic reconfiguration concept. The high degree of reusability together with the close interaction with the standard processor datapath allows the reconfigurable device to be of a smaller size than the FPGAs used in other approaches, where several entire kernels need to be mapped at the same time in the device.

On the other hand performance optimization with a reconfigurable processor requires a co-design tightly coupled between the processor core and the reconfigurable device. In order to avoid stalls and to improve the instruction level parallelism, the algorithm developer needs to manually tune the accelerated kernel 'C' code up to the boundaries defined by the application data dependencies. In this context, the user often needs to be conscious of the DFG abstraction that is implemented at the intermediate level by the Griffy-C description, and thus it is required to accurately handle the data flow across the reconfigurable device to obtain considerable performance improvements. It is important to notice that the optimization step is managed only at the DFG-level without any need for deep knowledge of the underlying architecture or circuit implementation. In

this context Griffy-C language also provides the user with a means to describe pipelines at a deep level of detail, though without needing skills in hardware design. For these reasons we state in the introduction that reconfigurable processors represent the natural extension of DSPs among other things for what concerns the algorithm development methodologies.

6.1 Reconfigurable software development time: hardware and software approaches

Depending on the constraints and on the type of algorithm, the implementation of an application on a reconfigurable processor can follow various different strategies, which lead to different trade-offs between performance and the development effort required. Hardware/software co-design for these architectures provides an additional dimension to the traditional design space for DSPs. Quality of service as well as real-time specifications can be used to select the implementation strategy and evaluate the development effort required.

Two main orthogonal factors can influence the time required to develop an application on reconfigurable processors:

- **Algorithm modifications** are applied. This is the case when the algorithm utilizes a description that is not well suited to the implementation on the target device or there is a different description which provides significant performance improvements. In the second case we include, for example, the description of non-standard operators (for example Galois Fields arithmetic) which achieve an important improvement when synthesized at the LUT-level, while a pure C code proves particularly inefficient. This approach is often referred as *hardware approach*, since it is orient to the optimization in a way similar to the optimized design of application-specific circuits.

- **Accurate scheduling** is applied. In this case the algorithm implementation is efficiently described by C language by means of high

level arithmetic/logic operations. This approach is also referred as *software approach*, since it is similar to the approaches used to optimize software programs, and the design space exploration performed at this point is:

- **clustering** basic operations to compose a Griffy operation, including the possibility of grouping clusters of concurrent or independent sub-graphs;
- **scheduling** the execution (i.e. through software pipelining) between the processor core and the reconfigurable device in an efficient way, thus avoiding stalls due to long latency instructions or memory accesses. However this is a common task that DSP developers are used to undertaking, in order to exploit dedicated functional units provided in the processor data-path.

Proficiency in co-design is the first key-point in order to obtain a reduction in the time-to-develop, given the expected performance. The time spent on application partitioning is often the dominant task, because this is an iterative task which includes analysis, description, validation and performance estimation. Acquired experience on application analysis for DSPs could help one early on to discard several solutions, thus focusing the design space exploration on a few significant options. Depending on both the ability of the user and the degree of modification that we want to introduce, the implementation process could take a long time, which is only justified if an appropriate performance improvement is achieved.

At a first glance, the programming of a reconfigurable processor could appear time-consuming as the ASIC design, requiring expertise on both the application and the architecture as well as non-common skills (partitioning, pipelining, and so on). However, compared to a traditional hardware design flow, the adopted development methodology does not require one to handle timing, critical paths, clock synchronization, or other hardware-specific features, which are extremely time-consuming in an HDL-to-silicon flow. In the case of Griffy, the framework in which the reconfigurable operations are described is strongly sequential, the working fre-

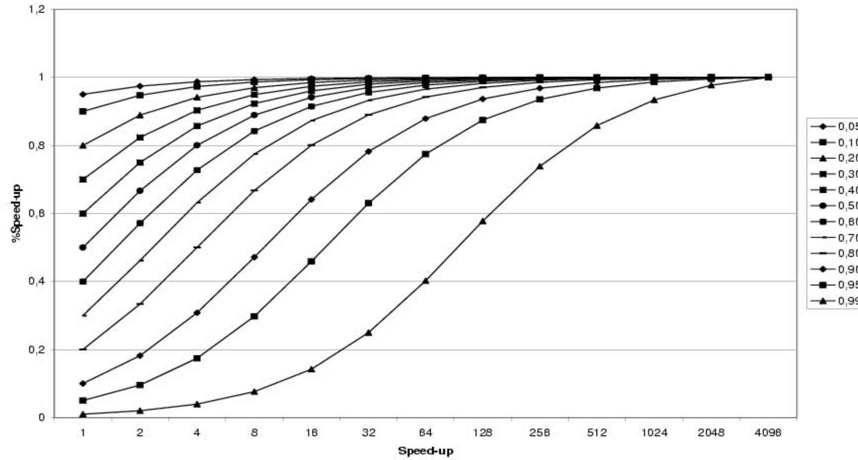


Figure 6.4: Variation of %speed-up wrt T_i and S_i

quency of a Griffy operation is assumed constant, since a reasonable worst-case condition is assumed (as in the case of PiCoGA), and performance improvements are achieved by managing the structure of the DFG pipeline through data dependencies directly described in the Griffy-C intermediate format. We can thus say that performance tailoring is achieved at the DFG level, simplifying the user-approach to the design-space.

In general talking, the approach of development shall be optimized in term of expected performance, kernel criticality and cost in term of development time. For example, long time development required for a hardware approach on a marginally import tasks, requiring $< 10\%$ of execution time, is probably not justified by a reasonable performance improvement. The impact of a i-th kernel optimization depend on the percentage of time T_i of the kernel with respect to the whole application and the local speed-up S_i by the formula:

$$Speed_up = \frac{1}{1-T_i(1-\frac{1}{S_i})}$$

When the local speed-up increase, the overall performance improvement depends only from the time T_i, thus giving a upper bound to the

performance. Fig. 6.4 show the percentage of overall speed-up gained with respect to the local speed-up, considering a set of different computational weight T_i. It is possible to see that the impact on overall performance saturate rapidly especially for kernel with $T_i < 50\%$.

For real application, the computational load is commonly distributed over N kernels, than the speed-up is determined by the formula:

$$Speed_up = \frac{1}{1-\sum_{i=0}^{i<N} T_i(1-\frac{1}{S_i})}$$

When a designer starts the implementation, he/she decides how many performance improvement is required to match the constraints and he/she needs to estimate how much development effort is necessary. It is necessary to optimize as much as possible all the critical kernels? Or, mixing hardware and software approaches it is possible to focus the development time only on few very critical kernels in order to achieve near-optimal performance? These are two important questions, for the engineering of reconfigurable system. To better explain the point, it could be considered the following example. Let us consider to have the 90% of the computational overload on 10 comparable kernels (9% per kernel).

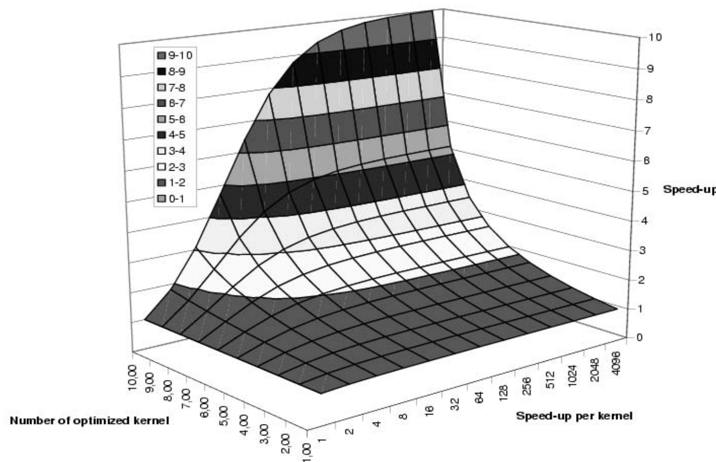


Figure 6.5: Variation of speed-up wrt #optimized kernel and local speed-up S_i

Fig. 6.5 shows a bi-dimensional space exploration obtained considering a variable number of optimized kernels and different (homogeneous) local speed-up. It is possible to see that better performance can be achieved with reduced local speed-up applied to all the kernels, with respect to the strong optimization of few parts. Moreover, usually, small local speed-ups could be obtained with a software approach, thus reducing the design time. This consideration could be extended to real cases, although the application-specific parameter could bias the final result and the choice of the best development strategy. For a quantitative analysis, please refer to the next chapter, where experimental results will be provided.

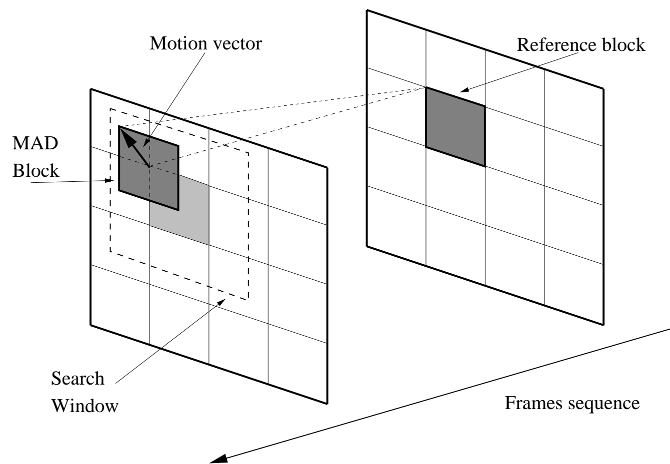


Figure 6.6: Motion estimation

6.2 Example of application mapping

6.2.1 MPEG-2 motion compensation on the XiRisc processor

Compression techniques are successfully employed in order to reduce the volume of transmitted data in audio/video communication devices. In particular, in the case of real video sequences (e.g. video conference), successive frames often have similar or identical content, mainly due to subjects or objects motion, thus introducing a high degree of correlation and a temporal redundancy that can be exploited using differential coding techniques[99]. MPEG video coding standards improve the compression scheme with a motion compensated prediction. Each frame is subdivided in blocks (or macro-blocks), of typically 16x16 pixels, and motion vectors are estimated searching among consecutive frames the block with a minimum distance. MPEG Software Simulation Group[100] proposes a public release of their MPEG-2 encoder compliant with ISO/IEC 13828-2[101] where motion estimation is computed using an exhaustive search pattern. The minimum absolute difference (MAD or L1 matching criteria) is determined among all blocks in a search window around the reference block (see Fig. 6.6).

Table 6.1: MPEG-2 computation-aware analysis

Algorithm	Clock Cycles	%
Motion Estimation	696144419	87%
Fast DCT	10820304	1.4%
Inverse DCT	14260740	1.8%
Prediction	6675078	0.8%
Quantization	12554590	1.5%
Inverse Quantization	9331982	1.3%
Variable-Length Coding	5260986	0.6%
Bitstream packing	30397513	3.8%
Communication among tasks	14445511	1.8%
Total	799891123	100%

This search approach, known as full-search motion estimation, allows the computation of an absolute minimum into the search window, but requires a very relevant computational cost. A computation-aware analysis of the MPEG-2 encoding engine, reported in Tab. 6.1 referring to a 12-frames sequence (or 1 group of pictures, GOP) with a resolution of 176x144 pixel (QCIF standard) and implemented on a VLIW RISC processor fetching 2-instruction per clock, shows how a largely dominant amount of computational time is spent over the motion estimation engine (about 90%), and specifically a significant contribution is due to the measurement of the distance between pairs of macro-blocks. This L1 matching criteria is a Sum of Absolute pixel-to-pixel Differences (or SAD) as described by the following formula:

$$Dist(u,v) = \sum_{i=0}^{i=N-1} \sum_{j=0}^{j=N-1} ||a_{i,j} - b_{i,j}||_{u,v} \tag{6.1}$$

Usual macro-blocks are squares of 16x16 pixel thus requiring 256 sums

of absolute differences repeated W^2 times, where W is the search window width in pixels. The dimension of the search window is thus defined by a trade-off between computational complexity and the compression factor. Standard general purpose embedded processors can hardly meet quality standards because of the described computational requirements. In such, several categories of multimedia processors have been proposed [102] in order to fill this efficiency gap. In fact, general purpose processors are inefficient when the Instruction Set Architecture (ISA) is not well suited to the task or when the amount of algorithmic parallelism in the application is greater than their capability to exploit it. On the contrary, Application Specific Integrated Circuits (ASICs) are optimized for the required task and offer excellent results in terms of speed and energy consumption, but their utilization implies non-recurring costs (NREs) that are often hardly justified by the application environment.

Reconfigurable computing appears a cost-effective and performing solution for data-intensive applications featuring deep pipelining and high concurrency. The key issue for the algorithm developer is the exploration of the design space in order to match application requirements with computational capabilities, thus determining the optimal partitioning of the task between hardware (the reconfigurable logic) and software (the processor core) resources. For example, multiplications or control-flow statements usually do not show particular advantages in space-based computations while bit-wise logic and concurrent multiple data arithmetic may offer significant improvement when implemented on an appropriately programmed FPGA-based device.

Full-search motion estimation performs an exhaustive comparison between all macro-blocks in the search windows and a reference block. A possible option to reduce complexity in the pixel-to-pixel distance measurement is to stop distance computation when the actual value exceeds the minimum pre-determined distance value. It is also possible to obtain a computational advantage choosing a suitable search path. In low motion sequences, such as in video conference environment, a locality criteria can be introduced in the search path performing a spiral path, as shown in Fig.

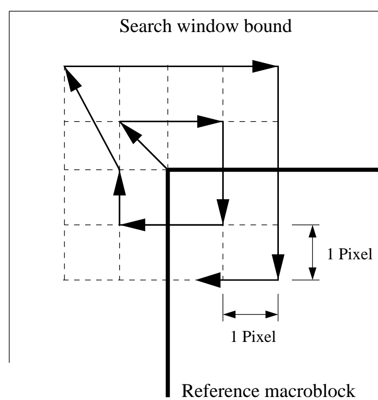


Figure 6.7: Search path

6.7.

This notwithstanding, the most significant algorithmic contribution in terms of complexity and time-cost considerations remains the computation of the distance between two given macro-blocks. Considering 16x16 pixel macro-blocks, the account of this distance requires 256 sums, 256 differences and 256 absolute value operations (ABSs). Using standard processors without an application-specific ABS hardware unit, this step can be performed using an "if-then-else" conditional structure and comparisons, severely increasing computational requirements.

The amount of assembly instructions spent for each distance measurement is around 1000, because the absolute difference computation performed through conditional statements requires roughly four instruction cycles, as is shown in the following assembly code (where r2, r3 are the current loaded pixels), repeated for all the 256 pixels.

```
        subu r4,r3,r2
        bgez r4,$L1
        subu r4,r0,r4
$L1:    addu r10,r10,r4
```

This computational kernel has been demonstrated as critical through a profiling-based analysis on the MPEG-2 encoding of a 12-frame sequence. In this test-case, the motion estimation phase accounts for about the 90% of the overall computation time, and more than 70% of that is spent on

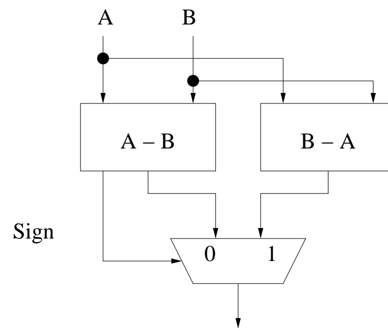


Figure 6.8: Absolute Difference (AD) DFG

the distance measurement function. In the case of the processor architecture, a first optimization step may be to use reconfigurable logic in order to implement the absolute difference (AD) operation. The space based computational pattern typical of hardware-oriented applications allows one to enhance the degree of parallelism in the computation using the graph shown in Fig. 6.8.

Since pixels are described using 8-bit unsigned variables, the area required for the implementation of this graph in a FPGA-like architecture or in the PiCoGA unit is extremely small. In fact, the Absolute Difference DFG mapped on the PiCoGA requires about 4% of the cells of the gate array. Still, the computational density achieved using this 8-bit pattern is small. Each processor datapath is used at 25% and the array under 4% of its potential computing power. The under-utilization of the PiCoGA

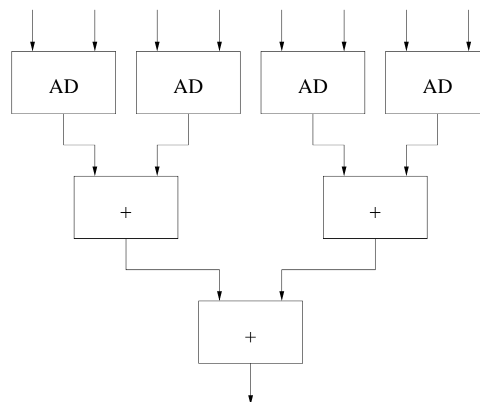


Figure 6.9: Concurrent 4-pixel Sum of Absolute Differences

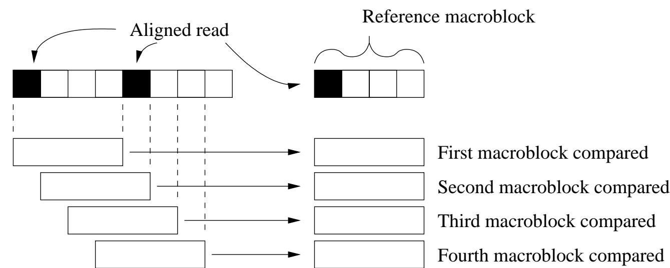


Figure 6.10: Memory layout

suggests an investigation of single-instruction-multiple-data (SIMD) computation patterns, implementing four concurrent absolute difference computations at a time. In this case, data transferred through the register file are integers and the array utilization goes up to 16%. It is also possible to embed the sum of these four absolute differences in the PiCoGA using a balanced logarithmic tree scheme, as illustrated in Fig. 6.9, thus maximizing instruction level parallelism and reducing both latency and issue delay of the graph. If the graph does not show dependencies across pipeline stages, the issue delay is minimal and then it is possible to overlap successive computations of long latency PiCoGA instructions significantly increasing the throughput.

A relevant problem introduced by the exploitation of concurrent computation over the gate-array, as is the case with the SAD proposed in Fig. 6.9, is the bottleneck caused from the number of accesses to system memory.

The XiRisc instruction set does not support misaligned memory access, so misaligned read/write operation from/to data memory is handled by byte-level load/store and packing/unpacking operations. As shown in Fig. 6.10, performing a spiral search path, only one frame over four is word-aligned, thus requiring four load-byte operations and byte packing (based on constant-step shift and bitwise-or) in order to build the correct inputs for the SAD operation. Operands'packing introduces a very significant overhead that significantly affects the possible speed-up figure. In conclusion, memory access bandwidth is the bottleneck which stops

an high throughput execution and thus the processor core introduces an upper-bound to achievable pipelining.

Increasing the PiCoGA input bandwidth performing on the gate array a sum of absolute differences which involves more than four pairs of pixels is a way that maximizes the PiCoGA area utilization, but it also increases the overhead introduced for packing operations due to misaligned memory access. In the next section we propose a modified implementation of the full-search motion estimation that avoids packing overhead and obtains significant performance figures following an alternative search path.

Improved Full-Search Motion Estimation

In the previous section we have shown how to improve the computation of the distance between two macro-blocks using PiCoGA. Unfortunately, the proposed implementation is affected by the problem of misaligned memory accesses that affect the available performance gain. In this section, an alternative approach will be proposed in order to improve full-search estimation avoiding the impact of operands packing, using well-known unfolding techniques in digital signal processing.

A macro-block in the search window is *aligned* if each 4-byte word in the 16x16 macro-block are aligned. In this case, the SAD shown in Fig. 6.9 can be computed without any overhead concurrently loading input data using only two memory accesses, as word-wise access is suitable for both reference macro-block and the current macro-block under comparison. Pixel-wise scan paths are unfortunately characterized by $\frac{3}{4}$ misaligned blocks. In order to overcome this problem, we have chosen to utilize a search path based on a 4-pixel-step spiral. We divided the search window in a Group of 4x4 Macro-blocks (GM) and we performed a spiral path among all the GMs in the search window. Each GM is thus internally parsed by rows, as depicted in Fig. 6.11.

Using this search pattern, each Group of Macro-blocks is aligned. It is then possible to perform concurrent computations of a row of GMs, thus nicely improving PiCoGA utilization, computational density and conse-

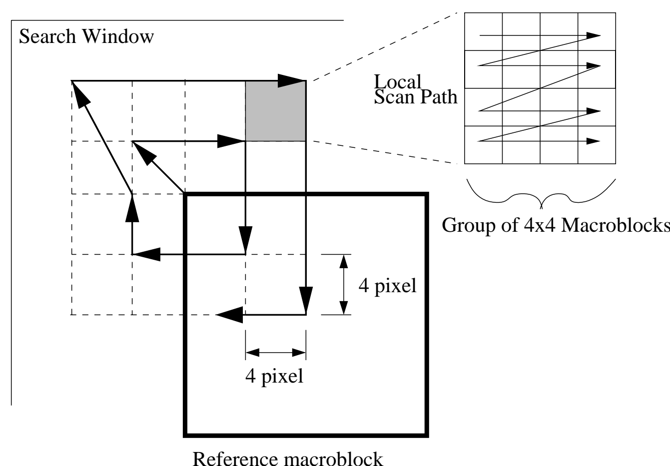


Figure 6.11: Enhanced search path

quently gaining performance. Four SAD operations on four bytes, as shown in Fig. 6.9, are used in order to implement a concurrent 4-blocks SAD operation that requires only word-aligned access to memory. In Fig. 6.12 the result of this unfolding approach applied both on the search path and on the local scan path is shown. The area required to implement this computational kernel is about 100% of the PiCoGA resources and the latency required in order to execute the SAD is 7 clock cycles which is the same of the concurrent 4-bytes SAD previously shown. We define this implementation as `sad4blk`.

The PiCoGA architecture is oriented at the pipelined elaboration of data-flow graphs in order to improve the throughput of a data-intensive computations. By overlapping successive executions of long latency PiCoGA instructions, it is often possible to improve the throughput up to bounds that are set by the issue delay on a side, and the processor-to-PiCoGA data bandwidth on the other. The main limitation comes again from the memory access bandwidth (three load versus one PiCoGA operation). But some further modifications of the inner loop allow one to achieve a higher degree of data reuse. For example, the reference block word can be reused for all 16 macro-blocks in the GM. The increment of the unfolding factor may also increase the number of registers statically allocated in order to store both temporary results and reusable data,

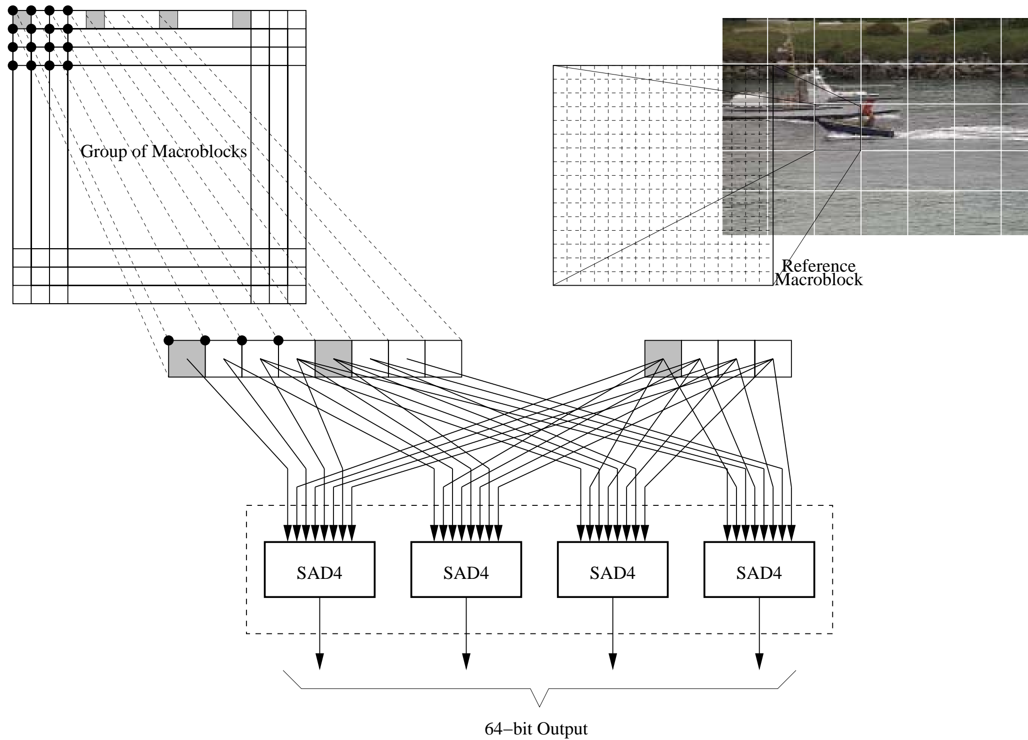


Figure 6.12: Concurrent 4-blocks SAD

introducing a critical trade-off between unfolding factor and register file occupancy. In order to avoid data dependencies that could lead to pipeline stalls, correlation among successive PiCoGA operations must be very small. This goal may be achieved by only taking the register file dimensions into careful account. The optimal unfolding factor is determined by the minimum number of stalls and the maximum usage of the register file without having to resort to the main memory for temporary variables storage.

We estimated as optimal trade-off the computation of 2-rows for each macro-block and the concurrent utilization of GM composed of 8 macroblocks each, leading to a loop utilizing 16 `sad4blk` operations. The pseudo-code in Fig. 6.13 shows this unfolded metric function. Intra-block unfolding decreases the number of used registers (for storing temporary results), but increases data dependencies. Thus, pipeline stalls needed to wait for PiCoGA writeback. Inversely, inter-block unfolding increases the

number of allocated registers, but decreases the number of processor stalls because of a low degree of correlation among macro-block distances. Using this unfolding factor, the instructions executed in the processor core, in order to provide data to the PiCoGA and to accumulate partial results, balances the PiCoGA operation latency avoiding stalls in the processor core thus allowing a good degree of pipeline utilization.

Furthermore, the register file usage, albeit scheduled with manual registers allocation, shows a high degree of coverage without increasing memory activity. The break mechanism previously introduced in order to stop the computation of distances greater than the actual minimum value shows a smaller impact, because of the unfolding factor applied in the inner loop. This reduction is more than compensated by the degree of utilization of the customized PiCoGA pipeline.

The unfolding technique applied to the spiral full-search path requires the concurrent availability of 4x4 macroblocks for each spiral step. This approach presents a drawback when the search path overcomes the boundaries of the search window or the frame size. In these cases, the chosen solution has been to read macroblocks exceeding the search window space discarding their distance values. This is possible with negligible computational cost. Each group of macroblocks is processed by reading 3-additional blocks with respect to the spiral path coordinates. By appropriately choosing the search window side, it is then possible to significantly reduce or to avoid altogether these boundary effects. In fact, boundary effects involve the control statements of the spiral-form path introducing additional check points which can be used in order to choose an appropriate measurement functions that involve or do not involve PiCoGA.

Fig. 6.14 shows the correspondent data flow graph, automatically generated through the C-based Place and Route flow described above. Each computational node represents an assembly-level operation, such as an addition or a subtraction, and is mapped over a set of logic cells. Nodes represented over the same line all belong to the same row and thus to the same pipeline stage. Consequently, the alignment shows the conformation of the pipeline stages (for example, in Fig. 6.14 is possible to iden-

```
for (j = macroblock_height; j > 0; j -= 2) {
    //lx: Frame width
    // First row of the corrent group of macroblocks - First internal row
    // RowPixels 0 - 3
    sad4blk(o1, o2, ((uint *) current_GM_row)[0], ((uint *) current_GM_row)[1], ((uint *) ref_macroblock)[0]);
    // RowPixels 4 - 7
    sad4blk(o3, o4, ((uint *) current_GM_row)[1], ((uint *) current_GM_row)[2], ((uint *) ref_macroblock)[1]);
    // RowPixels 8 - 11
    sad4blk(o5, o6, ((uint *) current_GM_row)[2], ((uint *) current_GM_row)[3], ((uint *) ref_macroblock)[2]);
    // RowPixels 12 - 15
    sad4blk(o7, o8, ((uint *) current_GM_row)[3], ((uint *) current_GM_row)[4], ((uint *) ref_macroblock)[3]);
    sad1_2 += o1 + o3 + o5 + o7; //Concurrent two 16-bit add
    sad3_4 += o2 + o4 + o6 + o8; //Concurrent two 16-bit add
    // First row of the current group of macroblocks - Second internal row
    current_GM_row += lx;
    sad4blk(o1, o2, ((uint *) current_GM_row)[0], ((uint *) current_GM_row)[1], ((uint *) ref_macroblock)[0]);
    sad4blk(o3, o4, ((uint *) current_GM_row)[1], ((uint *) current_GM_row)[2], ((uint *) ref_macroblock)[1]);
    sad4blk(o5, o6, ((uint *) current_GM_row)[2], ((uint *) current_GM_row)[3], ((uint *) ref_macroblock)[2]);
    sad4blk(o7, o8, ((uint *) current_GM_row)[3], ((uint *) current_GM_row)[4], ((uint *) ref_macroblock)[3]);
    sad5_6 += o1 + o3 + o5 + o7; //Concurrent two 16-bit add
    sad7_8 += o2 + o4 + o6 + o8; //Concurrent two 16-bit add
    // Second row of the current group of macroblocks - First internal row
    current_GM_row -= lx; ref_macroblock += lx;
    sad4blk(o1, o2, ((uint *) current_GM_row)[0], ((uint *) current_GM_row)[1], ((uint *) ref_macroblock)[0]);
    sad4blk(o3, o4, ((uint *) current_GM_row)[1], ((uint *) current_GM_row)[2], ((uint *) ref_macroblock)[1]);
    sad4blk(o5, o6, ((uint *) current_GM_row)[2], ((uint *) current_GM_row)[3], ((uint *) ref_macroblock)[2]);
    sad4blk(o7, o8, ((uint *) current_GM_row)[3], ((uint *) current_GM_row)[4], ((uint *) ref_macroblock)[3]);
    sad1_2 += o1 + o3 + o5 + o7; //Concurrent two 16-bit add
    sad3_4 += o2 + o4 + o6 + o8; //Concurrent two 16-bit add
    // Second row of the current group of macroblocks - Second internal row
    current_GM_row += lx;
    sad4blk(o1, o2, ((uint *) current_GM_row)[0], ((uint *) current_GM_row)[1], ((uint *) ref_macroblock)[0]);
    sad4blk(o3, o4, ((uint *) current_GM_row)[1], ((uint *) current_GM_row)[2], ((uint *) ref_macroblock)[1]);
    sad4blk(o5, o6, ((uint *) current_GM_row)[2], ((uint *) current_GM_row)[3], ((uint *) ref_macroblock)[2]);
    sad4blk(o7, o8, ((uint *) current_GM_row)[3], ((uint *) current_GM_row)[4], ((uint *) ref_macroblock)[3]);
    sad5_6 += o1 + o3 + o5 + o7; //Concurrent two 16-bit add
    sad7_8 += o2 + o4 + o6 + o8; //Concurrent two 16-bit add
}
```

Figure 6.13: Unfolded SAD function based on `sad4blk`

tify 5 pipeline stages). Some operations, such as constant-step shifts employed for word-wise to byte-wise pixel unpacking, can be made using only the programmable interconnections of the gate array reducing both area and latency of the graph. These instructions do not occupy any stage in the hardware pipeline, thus are defined *routing-only* operations and are

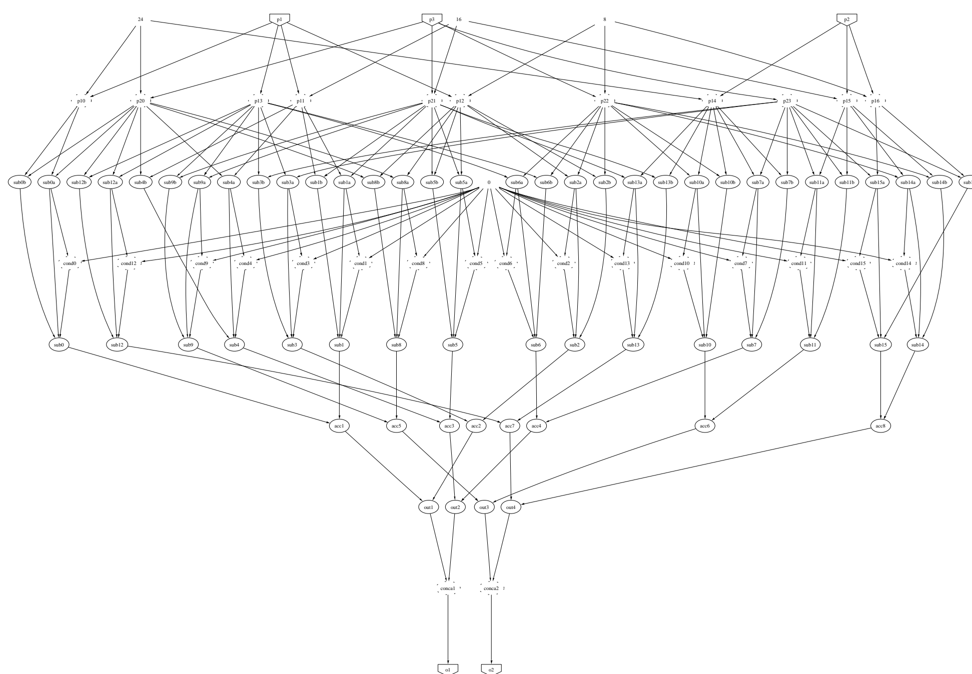


Figure 6.14: `sad4blk` DFG

drawn, in the DFG, using dotted nodes. Fig. 6.15 depicts the `sad4blk` instruction mapped over the PiCoGA.

It should be observed that `sad4blk` cannot be effectively used in the case of the half-pel refinement of the motion vector. Half-pel precision is utilized in MPEG compression in order to reduce the residual error in the differential coding, but the interpolation among adjacent macroblocks and the compile-time non-predictable alignment of the minimum distance macroblock requires memory accesses to be performed at byte level. The need of a packing step in order to feed PiCoGA with an appropriate workload would require such relevant processor activity to vanify any advantage introduced by configurable computation. For this reason, in our implementation, the half-pel refinement phase is performed through processor-only computation.

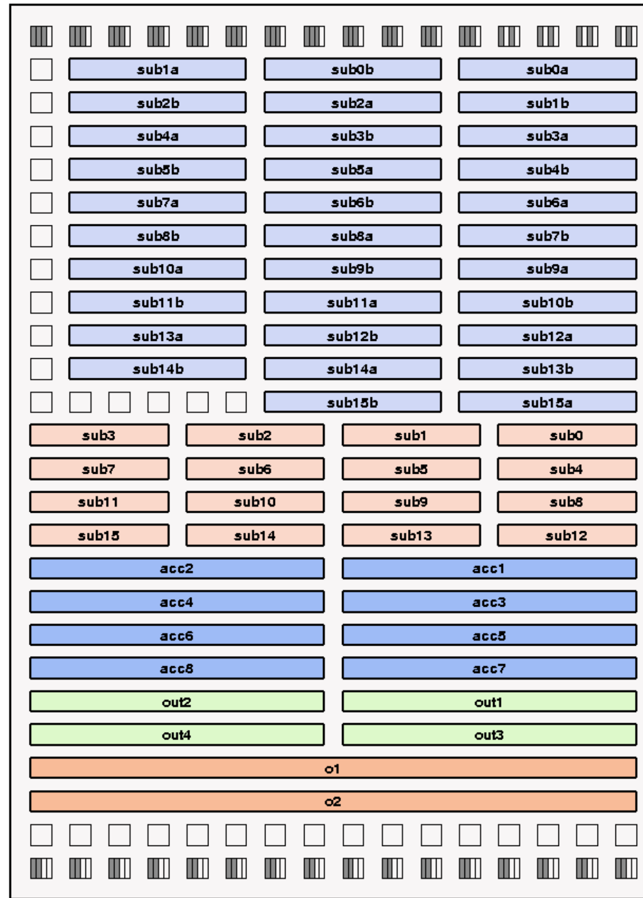


Figure 6.15: `sad4blk` Place & Route

Performance Evaluation

An evaluation of the effectiveness of the implemented solution can be obtained comparing results achieved using the XiRisc reconfigurable processor with the performances obtained by a general purpose embedded VLIW RISC processor not augmented by PiCoGA. Operating frequency and function units availability are the same in the two cases in order to have a fair proof of the performance enhancements bound to the instruction set architecture metamorphosis.

The synthesizable HDL model of the processor can be used in order to verify the correctness of the implementation, but the huge simulation time required by MPEG compression algorithm would not allow an exhaustive

analysis over a significant benchmark. Faster simulations can be obtained using an instruction set simulator (ISS) which describes the functionality of the processor. Depending on the desired level of accuracy it is possible to use *bit-accurate* or a *cycle-accurate* simulation.

Instruction-accurate simulators, such as the internal ISS of the GNU GDB debugger, performs an evaluation of the computation without considering pipeline stalls or memory waits. On the other hand, cycle-accurate simulators such as ISS generated from LISA (Language for Instruction Set Architecture) evaluate accurately pipeline stalls, and can be embedded into a System-C environment in order to evaluate memory hierarchy impact. For a qualitative performance evaluation we have used a profiling tools based on the GNU-GDB ISS, that is be used to provide a program trace (a trace file that annotates all computed instructions) estimating only the processor stalls introduced by the PiCoGA register lock mechanism. The results of the profiling are then back-annotated on C and assembler source code.

As described in [66], the energy consumption of the XiRisc processor architecture can be roughly considered proportional to the number of memory accesses, which in turn is mainly due to instruction fetches. By collapsing a set of assembly instructions in a single instruction that triggers PiCoGA elaboration, it is possible to reduce the number of fetches and thus to decrease energy consumption. Of course, this decrease is traded with the overhead in power consumption due to the reconfigurable hardware unit (leakage power) and the elaboration of the unit itself (dynamic power):

- the first component is proportional to gate array area;
- the second one is a dynamic component of energy consumption due to PiCoGA elaboration and depends from both the input data and the DFG implemented.

This simplified model was used in order to estimate energy consumption on the traces provided by software simulation. The model was empirically verified from measurement performed on silicon prototypes.

Table 6.2: Test-sequence features

Sequence title :	Coast-guard
Number of frames:	12 (1 GOP)
Frame standard :	QCIF (176x144)
YUV standard :	4:2:0
Interleaving :	No
Macroblocks :	16x16
Search windows :	16x16 (-8,+7)

The effectiveness of the introduced motion estimation implementation has been evaluated on a test sequence composed by a group of 12 frames in QCIF standard. Encoding this sequence (Table 6.2), we have analyzed an entire Group of Picture (or GOP) featuring both backward and forward predictions. By extracting from the MPEG profiling analysis the results referred to motion estimation it is possible to observe a performance improvement up to one order of magnitude (in the case of full-pel precision) comparing XiRisc with a general purpose VLIW RISC processor.

Results are shown in Table 6.3, where the “distance” speed-up (about 18x) is reported referred to the kernel directly involved in the PiCoGA-driven computation and the motion estimation performance is referred both at the case of full-pel and half-pel precision. These performances are relevant also in the case of the motion estimation algorithm with half-pel analysis which introduces a significant overhead due to control flow statements and spiral path handling that justify the speed-up figure decrement.

The performance gain achieved using `sad4blk` depends on the required search area. As explained in [104], it is necessary to find a trade-off between search area, compression factor and computation-time. Depending on the amount of available computation, several algorithms feature a reduced computational requirement inspecting a subset of checkpoints through hierarchical paths or through search windows with variable side. In Fig. 6.16, considering the full-search engine (with full-pel precision),

Table 6.3: Performances

	Speed-up	Energy Consumption Reduction
Distance	18x	-
M.E. Full-Pel	10x	80%
M.E. Half-Pel	7x	75%

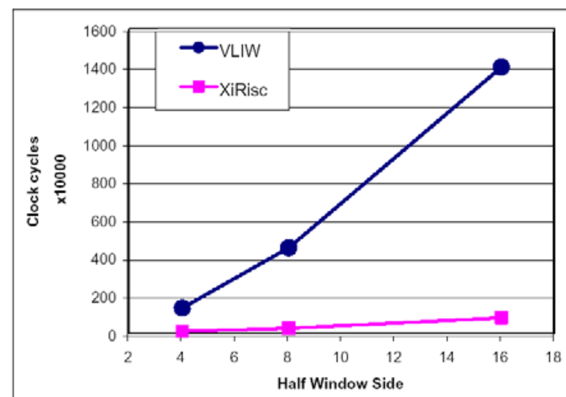


Figure 6.16: Full-Search workload vs. search window side

we show a linear increase of computational workload proportionally to the search window sides comparing a VLIW processor with a XiRisc implementing the `sad4blk`.

Even if the computational workload is small, this XiRisc configuration can be effectively used obtaining a significant gain. In the border-case represented by fast motion estimation algorithms (e.g. the algorithms overviewed in [105]), where the matching criterion is applied to a very small number of distributed macroblocks, `sad4blk` can be used in order to avoid computational overheads due to data misalignments achieving a speedup that can be estimated about 5x in full-pel distance measurement. The impact on the overall MPEG-2 encoding is summarized in Table 6.4.

Using the power estimation model described in the previous section, an energy consumption reduction of about 75-80% has been achieved, mainly due to PiCoGA intensive usage in the most significant computa-

Table 6.4: MPEG-2: final results

Algorithm	Clock Cycles	%
Motion Estimation	115671891	53%
Fast DCT	10820304	4.9%
Inverse DCT	14260740	6.4%
Prediction	6675078	3%
Quantization	12554590	5.7%
Inverse Quantization	9331982	4.3%
Variable-Length Coding	5260986	2.3%
Bitstream packing	30397513	13.9%
Communication among tasks	14445511	6.5%
Total	219418595	100%

tional kernel thus demonstrating the effectiveness of reconfigurable computing approach for energy-critical applications.

6.2.2 AES/Rijndael implementation on the DREAM adaptive DSP

Security of data is becoming an important challenge for a wide spectrum of applications, including communication systems (with high privacy requirements), secure storage supports, digital video recorders, smart cards, cellular phones. Resistance against known attacks is one of the main properties that an encryption algorithm needs to provide. When a new attack is demonstrated as effective (also in term of computation time), the update of the encryption system is a real necessity to guarantee the security of data.

In November 2001, the National Institute of Standard Technology (NIST) announced the Advanced Encryption Standard (AES) [106], as a replacement of the Data Encryption Standard (DES). The Rijndael algorithm [107], selected among 15 candidates, is a symmetric key algorithm based on a substitution-permutation network, where most of the calculations are done using Galois Field (GF) arithmetic defined over the field GF(2^8) with the *irreducible polynomial* $x^8+x^4+x^3+x+1$.

Applications requiring high performance and/or low power consumption are today implemented using dedicated hardware accelerators with the downside of higher development costs and lack of flexibility (i.e. algorithm update or parameter changes) with respect to software implementations. In this context, reconfigurable hardware such as Field Programmable Gate Arrays (FPGAs) seems to bridge the gap between performance and flexibility required to guarantee the necessary updates. For complex System-on-Chip, where the area budget dedicated to a single computational island is a constraint, reconfigurable architectures (RAs) for embedded applications were proposed as hardware accelerators, including embedded FPGAs, reconfigurable processors and reconfigurable data-paths.

In this section, an implementation of the AES/Rijndael algorithm on the DREAM architecture will be presented. The DREAM architecture is composed of a reconfigurable data-path (the 3rd generation Pipelined Configurable Gate Array, or PiCoGA-III) controlled by a 32-bit RISC processor.

PiCoGA-III is directly interfaced to a high-bandwidth memory sub-system through programmable address generators, featuring for example vectorized and modulo addressing. An important key point is that the PiCoGA-III features a native support for operations in GF(2^4), thus allowing easy and effective implementations of composite fields that provide the mathematical back-ground for many applications, including Reed-Solomon Codes.

Overview: the AES/Rijndael algorithm

The Rijndael algorithm [107] is a symmetric key cipher implementing a substitution-permutation network. The size of both ciphered block and key depends on the security level required, as well as the number of iterations (rounds) required to encrypt the plain-text. As an example, the U.S. Government requires 128-bit keys for SECRET data, while the TOP-SECRET level requires 196 and 256-bit keys. While Rijndael supports a large range of block and key sizes, the NIST standardized a subset of them, using only 128-bit blocks and 128, 196 and 256-bit keys [106]. For ciphering a stream, AES/Rijndael can be applied in many schemes, including ECB (Electronic Codebook) and CBC (Cipher Block Chaining) [108]. While the EBC mode ciphers each block independently to the other ones, the CBC XORs the plain-text with the previously ciphered block, preventing the coding of equal plain-blocks with equal ciphered-blocks. On one hand, the CBC mode introduces an additional level of security *wrt* EBC, but on the other hand we have an additional feedback that limit the peak performance, especially for hardware implementation.

The encryption process starts arranging the block in a matrix form termed *State*. Let us consider as reference the 128-bit (block and key) Rijndael. In this case, the *State* (`S`) is a 4×4 array of bytes in which the 128-bit block is arranged by rows. The *State* is thus encrypted by the iterative application of 4 operations, as described in the following pseudo-code.

```
S=in; Nb = 128;
S=AddRoundKey(S, key[0,Nb-1]);
for (i=1; i<Nround; i++) {
  S = SubBytes(S);
  S = ShiftRows(S);
  S = MixColumns(S);
  S = AddRoundKey(S,key[i*Nb,(i+1)*Nb]);
}
S = SubBytes(S);
S = ShiftRows(S);
S = AddRoundKey(S,key[i*Nb,(i+1)*Nb]);
out = S;
```

The number of iteration (Round) depend on the key size, and ranges from 10 to 14. Four basic operations are applied to the *State*:

SubBytes: is a non-linear substitution step applied to each byte of the *State* array, that is substituted with its inverse multiplicative over GF(2^8). Then, an affine transformation ($a' = M \times a + c$) is applied, as described by the following equivalent equation:

$$a'_i = a_i + a_{(i+4)\ mod\ 8} + a_{(i+5)\ mod\ 8} + \quad (6.2)$$

$$a_{(i+6)\ mod\ 8} + a_{(i+7)\ mod\ 8} + c_i$$

where a' and a are bytes of the *State* array, c is the vector (01100011). The non-linear substitution applied to each byte is also known as S-Box.

ShiftRows: operates on the rows of the *State*, rotating them to the left by a shift step equal to the row index.

$$A'_{i,j} = A_{(4+i-j)mod4,j} \quad (6.3)$$

where i and j are respectively the column and row indexes.

MixColumns: operates on the four bytes of each column of the *State* array, that are treated as the coefficient of a 4-th order polynomial over GF(2^8). The MixColumns step performs a multiplication (modulo $x^4 + 1$) with the fixed polynomial $3x^3 + x^2 + x + 2$.

AddRoundKey: represents the last operation of each Round and performs an addition over GF(2^8) between the *State* and the *Round Key*, a 4×4 array generated from the original key by an expansion step in order to provide different key-words to different rounds.

The key expansion step, also known as *Key Schedule*, is performed before the encryption, and is described with mathematical operations, mainly based on the application of S-Box and word rotation [106, 107].

All the operations previously described are invertible in a very straightforward manner, resulting a decoding schema very similar to the encoding one. In particular, the computational complexity is more or less the same, since the kind of applied operations is the same.

The Advanced Encryption Standard implemented by the Rijndael algorithm can be efficiently implemented in both software and hardware. 8-bit processors can directly implement most of the operations required by AES since they are natively working on 8-bit variables (e.g. ShiftRows, AddRoundKey and MixColumns), while the S-Box is more efficiently implemented using a 256-entry 8-bit hash table. 32-bit processors implement fast Rijndael combining the different step of a round transformation in a single set of hash-tables. As a result, 4 tables with 256 32-bit values (termed T-Box) substitute most of the round operations, leaving to the dynamic computation XORs and rotations [107]. Comparing this optimized version with the basic one, about one order of magnitude in performance is gained on a RISC processor. Implementations on TI DSPs are discussed in [112]: a 112.3 Mbit/sec throughput (@ 200MHz) is achieved on the C62x architecture for the encoder, 1.6× faster than a Pentium-Pro working at the same frequency. Moreover, instruction set extensions dedicated to Rijndael are present in the literature, such as [109, 110].

Hardware implementations of AES are optimized by the exploitation of the available parallelism. Hence, the design of hardware accelerators for AES begins from the 1-to-1 unfolding of the Round definition, as shown in Figure 6.17. For the ECB mode, the Rijndael algorithm can be completely unrolled and pipelined, thus improving the available throughput up to the technological limit. The undeniable drawback is the consider-

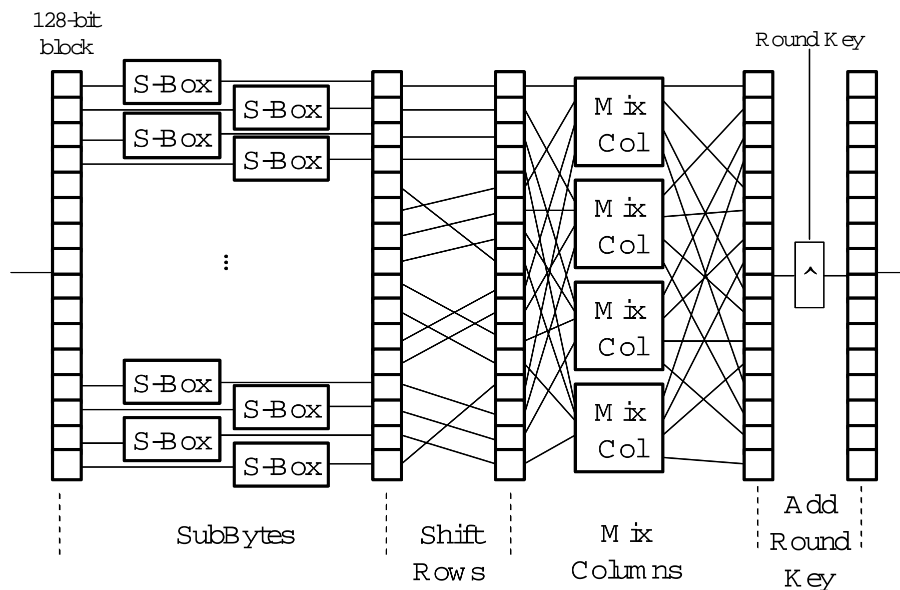


Figure 6.17: Common AES-Round block diagram

able augment in area occupation. Examples of AES implementations for stand-alone FPGAs are [115, 116, 113, 117, 114], providing 2-30 GBit/sec throughout. Hybrid solutions, coupling a processor with FPGA technology, are implemented in the Xilinx Virtex II Pro platform [114, 118], achieving performance up to 1.2 GBit/sec. For embedded applications, where the area budget is a constraint, devices with restricted size are proposed. Embedded FPGAs (e.g. [111]) are the most direct "translation" of the traditional field-programmable technology to the market of IPs suitable for SoC integration. Alternatively, and depending on the application field, reconfigurable data-paths (e.g. [53, 55]) are used as hardware-programmable accelerators. As an example, in [120] a reconfigurable datapath challenges a set of cryptographic applications.

Implementation of basic GF(2^8) operations

An important property of Galois Fields is that they are univocally defined by the number of elements. What can be changed, depending on the *irreducible polynomial*, is the representation. Therefore, the GFs are isomorphic with respect to an *irreducible polynomial* change and a transformation ma-

trix can be defined in order to change the representation. As described in Paar' PhD Thesis [121], this implies that GF(2^8) can be seen as a composite field GF($(2^4)^2$) whose elements are represented by 1-order polynomials $\alpha x + \beta$ with $\alpha, \beta \in$ GF(2^4). PiCoGA-III features a native support of GF(2^4) with the ***irreducible polynomial*** $x^4 + x + 1$. This means that each RLC can be programmed to perform both the ***sum*** ($\oplus$) operation, implemented by LUT as a 4-bit XOR, and the ***multiplication*** ($\otimes$) operation, implemented by the dedicated GF multiplier.

The AES/Rijndael algorithm requires to implement three operations on GF(2^8): the ***sum***, the multiplication by constant amount, and the inverse multiplicative. While the ***sum*** and the multiplication with constant amount can be described (in Griffy-C) and implemented (on the PiCoGA) with standard C (XORs, ANDs and shifts), the implementation of the inverse multiplicative over GF(2^8) benefits from the GF capabilities of PiCoGA-III. By definition [122], the inverse multiplicative on the composite field GF($(2^4)^2$) (using the ***irreducible*** $x^2 + x + \omega_{14}$) is:

$$\begin{aligned} (\alpha x + \beta)^{-1} &= \alpha \otimes \theta^{-1} x + (\alpha \oplus \beta)\theta^{-1} \\ \theta &= \alpha^2 \otimes \omega_{14} \oplus \alpha \otimes \beta \oplus \beta^2 \end{aligned} \tag{6.4}$$

Figure 6.18(a) shows the straightforward implementation of the inverse multiplicative obtained from equation (6.4). Basic blocks are aligned per pipeline stage, and each basic block can be mapped on one RLC (the inverse on GF(2^4) is a 4-in 4-out function implemented by LUT). The full retiming, needed to maximize the throughput, requires 7 additional registers (dashed-line blocks), for a total of 17 RLCs distributed over 5 rows. Figure 6.18(b) shows an optimized inverse multiplicative generated by rewriting the equation (6.4) in the following form:

$$\begin{aligned} (\alpha x + \beta)^{-1} &= (\alpha^{-1} \otimes \delta)^{-1} x + ((\alpha \oplus \beta)^{-1} \otimes \delta)^{-1} \\ \delta &= \alpha^2 \otimes \omega_{14} \oplus \beta \otimes (\alpha \oplus \beta) \end{aligned} \tag{6.5}$$

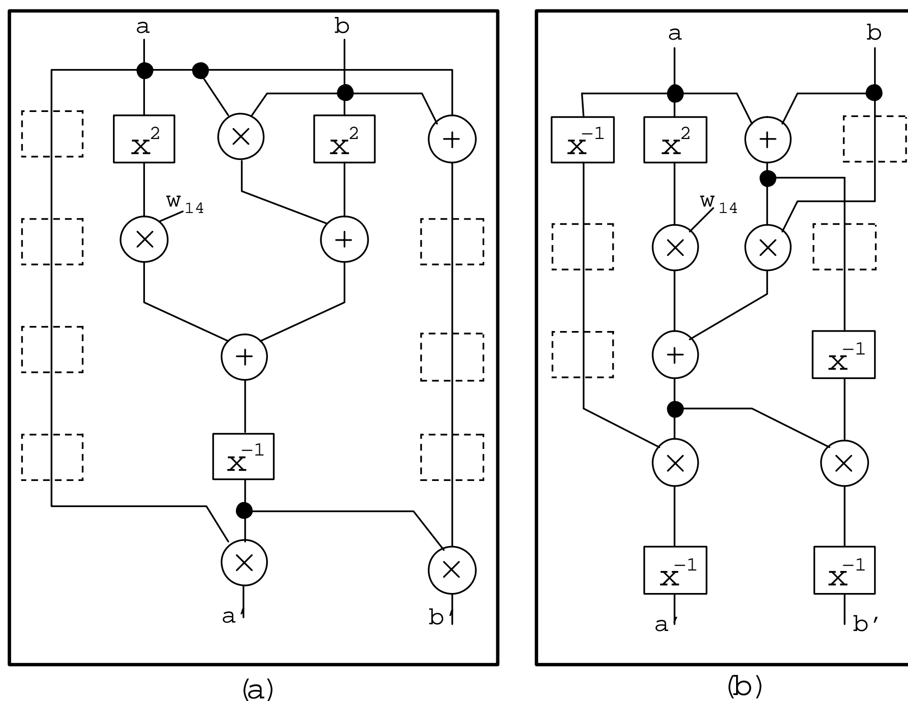


Figure 6.18: Inverse multiplicative on composite fields schemes

In this second case we have an issue-delay of 2 cycles, requiring only 4 additional registers (for a total of 15 RLCs) for the full retiming. The max-width of this implementation schema is 4 RLCs, allowing a better packing of multiple instances of the inverse multiplier in the PiCoGA rows (each of them composed by 16 RLCs). To complete an S-Box, we need to add the isomorphism matrix and the successive affine transformation. Two rows with respectively 4 and 2 RLCs are required for the input isomorphism, while the output isomorphism and the affine transformation can be collapsed together, with the same resources occupation (4+2 RLCs).

Implementation of AES/Rijndael

A goal of our AES/Rijndael implementation is to be flexible for both block and key size. Hence, we have analyzed, in relation with DREAM capabilities, the following properties of Rijndael algorithm. First of all, since the SubBytes operation does not depend on the position of each byte, the ShiftRows can be performed before the SubBytes. In addition, ShiftRows

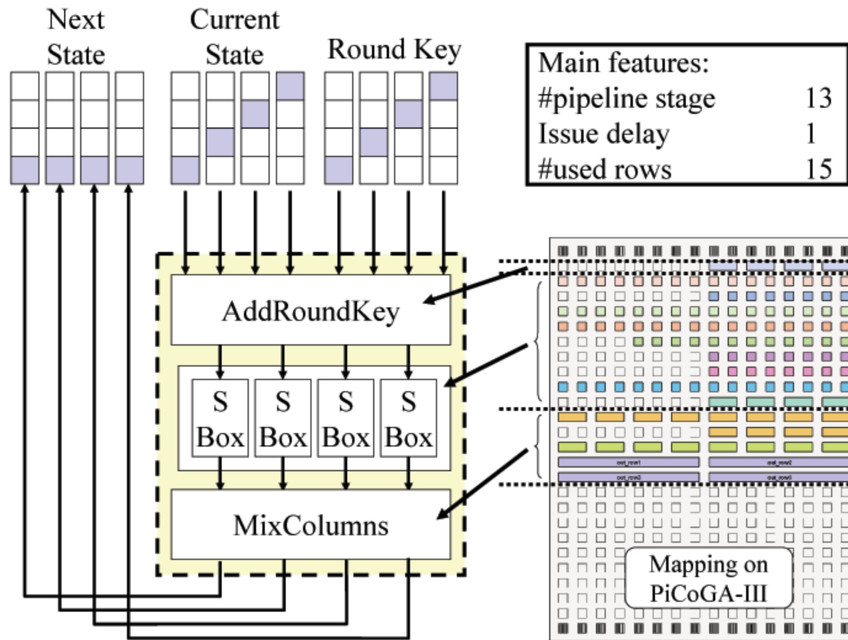


Figure 6.19: AES/Rijndael selected kernel and implementation

performs a rotation which can be implemented using modulo addressing. Hence, using different memory banks for storing the different rows of the *State* matrix, PiCoGA is able to load a new *State* column for each cycle. The rotation applied by ShiftRows is handled by changing the starting address of each bank, while the different number of columns (for the generic Rijndael) is handled by setting the address generator end-of-count. The organization by column allows the packing of the MixColumns function in the same PiCoGA operations.

Figure 6.19 shows the corresponding implementation scheme. This PGAOP performs AddRoundKey, SubBytes and MixColumns for the 4 bytes in a column concurrently, leaving the addressing engine to handle the ShiftRows for both block and key access. A different set of buffers is used to store PGAOP results, since it is not possible to read-and-write a memory bank in the same cycle. This implementation requires 4 PGAOP call in order to accomplish one AES/Rijndael Round, after that we need

	Clock cycles per 1 block		
block/key size	Scalable Version	Optimized Version	Key Expansion
128/128	408	285	192
128/192	466	329	216
128/256	524	373	240
256/128	455	-	319
256/192	521	-	367
256/256	587	-	415

Table 6.5: AES/Rijndael encoder performance

to re-configure the interconnect cross-bar in order to swap the used I/O buffers. Although this operation could be performed in parallel to the PGAOP computation (destination port are stored internally to the PiCoGA during the PGAOP triggering), this reconfiguration break the best pipeline evolution. For the EBC mode, there is not dependency among the encryption of successive blocks, thus it is possible to interleave the encryption of more than one block in order to mitigate the impact of the interconnect reconfiguration. The stride factor allows the address generator to jump to the next block when the Round is finished. The last Round requires the implementation of a dedicated PGAOP, without MixColumns and within an additional AddRoundKey before the SubBytes needed by the loop transformation introduced before. Only 11 pipeline stages are required for this goal, but the area occupation is increased to 17 rows because of an unfavorable requirement of additional retiming registers necessary to maintain the issue-delay equal to 1.

For 128-bit block only, the PiCoGA-III is able to output a whole 4x4 block, then it is possible to implement an optimized PGAOP using only the simple registers. When blocks interleaving is not applicable (e.g. in CBC mode), we can achieve a further 1.4× speed-up reducing the configuration overhead, through the utilization of simple registers instead of address generators to exchange data with the PiCoGA. Two additional

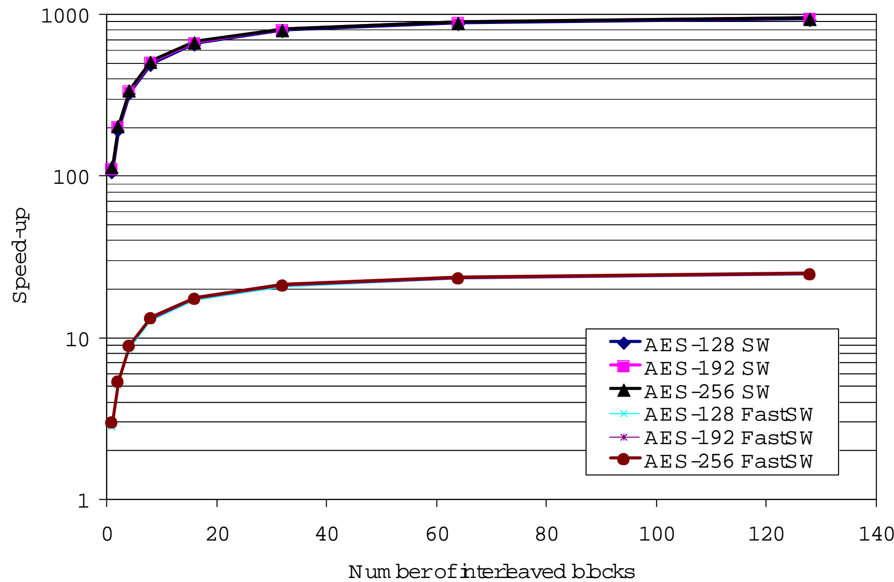


Figure 6.20: Speed-ups *wrt* RISC processor

shift registers (and the corresponding control logic) shall be mapped on the PGAOP because the ShiftRows requires to be implemented internally. Data are loaded at the first PGAOP trigger, while other 3 three additional triggers are required to provide the correct result.

Experimental results and comparisons

We have implemented the AES/Rijndael algorithm on the DREAM cycle-accurate Instruction Set Simulator (ISS), based on CoWare technology. The RISC processor is modelled using LISA language, while the memory sub-system and the PiCoGA are modelled using a mix of SystemC and C/C++. Frequency and power consumption figures are estimated starting from measurement on the silicon prototype in [75], featuring a comparable design complexity. Both scalable and optimized implementations presented in the previous section were considered in our analysis and the cycle count obtained is reported in Table 6.5. Results are provided for the encryption of a single block, considering various block and key sizes. At the frequency of 200MHz, it is possible to achieve a throughput up to 90Mbit/sec using a scheme applicable in both EBC and CBC modes.

In EBC mode, the scalable solution can interleave the encryption of more than one blocks, exploiting as much as possible the computational efficiency of DREAM. Pipelining the computation on the PiCoGA-III, the obtained speed-up figures raises from 100$\times$ to 930$\times$ *wrt* the ANSI-C Reference Code (v. 2.2) running on a RISC processor at the same frequency, while it raises from 3$\times$ to 24$\times$ *wrt* a fast software implementation (by C. Devine, on-line available at the Rijndael Home Page [107]) working on the same RISC processor. Figure 6.20 shows the achieved speed-ups *versus* the level of interleaving applied, hence in relation to the number of block concurrently elaborated.

Figure 6.21 shows an analysis of the throughput with respect to the interleaving factor applied. As a consequence, ciphering 64 or 128 blocks, the benefit of pipelining the computation inside the PiCoGA-III mitigates the overhead due to interconnect configuration changes, allowing one to obtain up to 546 Mbit/sec of throughput. Considering the case of AES-128, the throughput increases from 63 to 546 Mbit/sec in a way that is proportional to the average number of active rows inside the PiCoGA. In fact, the average number of active rows growth from 1.5 rows/cycle to 12.8 rows/cycle, respectively corresponding to 10% and 85% of the PGAOP. With 256-bit block size, the memory utilization growths faster, then the 128-block interleaving cannot be applied.

Comparisons with other AES-128 implementations are reported in Table 6.6, including both fast software (with an assembly hand-coded Pentium-III) and hardware approaches. Furthermore, a processor with custom-designed ISA [109] is considered too. For the hardware approaches, we have taken into account folded schemes implemented on both FPGA and ASIC (0.18μm) prototype. The energy efficiency (Mbit/sec/mW) shows the density advantage of DREAM with respect to the other “programmable” solutions. For this purpose, the power consumption of DREAM is estimated in a range from 80 mW (CBC) to 180 mW (EBC), depending on the different PiCoGA-III utilization and correlated memory activity.

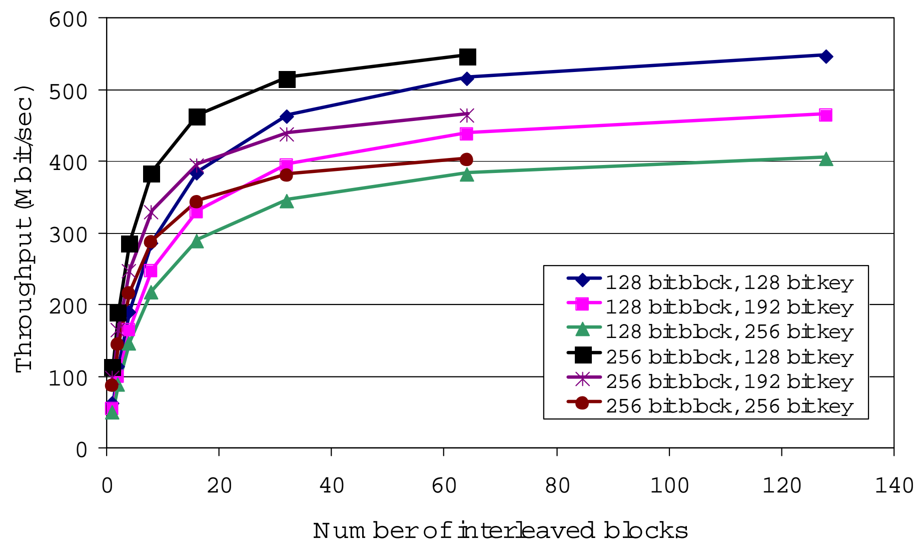


Figure 6.21: Throughput vs. interleaving factor

	Frequency MHz	Throughput Mbit/sec	Energy eff. Mbit/sec/mW
DREAM (EBC)	200	546	3.03
DREAM (CBC)	200	90	1.12
ARM9(1) [123]	500	46.6	0.32(1)
ARM9(2) [123]	250	23.3	0.67(2)
TI C62x [112]	200	112	n/a
Pentium-III [119]	1130	645	0.015
Ravi [109]	188	17.2	n/a
Lu [118]	196	1197	n/a
Chaves [114]	100	1258	n/a
Sch. [119] FPGA	77	640	0.39
Sch. [119] ASIC	154	1280	22.8

(1) ARM926EJ-S Speed-Opt. 90nm 0.29 mW/MHz (www.arm.com)

(2) ARM926EJ-S Area-Opt. 90nm 0.14 mW/MHz (www.arm.com)

Table 6.6: AES-128 encryption comparisons

6.2.3 Low-complexity transform for H.264 video encoding

The H.264 video encoding architecture [124] has many innovations if compared to previous standards and provides a compression gain of 1.5-2.0× over in relation to them. Among the other things, this standard introduces:

- a new low-complexity transform and quantization approaches [125] employing only integer arithmetic without multiplications. Its coefficients and scaling factors can be elaborated using a 16 bit arithmetic, leading to a significant complexity reduction, and allowing a more efficient hardware implementation, in particular for reconfigurable devices.

- new cost functions for the definition of the macro-block distance metric in motion compensation heuristics. In particular two metrics are defined, the sum of absolute difference (SAD) and sum of absolute transform difference (SATD) based on Hadamard transform. Motion compensation is also used for the efficient intra-frame encoding, by the introduction of nine motion mode. Intraframe prediction can be used to encode very efficiently also static images, with the same signal to noise ratio of JPEG2000, but with a better compression factor [126].

This section and the next one will introduce the implementation of these critical kernels on the PiCoGA-III reconfigurable device in the DREAM adaptive DSP. It will be also illustrated the techniques of optimization adopted to obtain an optimal implementation of these computational kernels.

H.264 Transform

The structure of H.264 imposes several requirements on the design of residual coding. In traditional video encoding standard, residual decoding contains the possibility of drift (mismatch between the decoded data in the encoder and decoder). The drift arises from the fact that the inverse

transform is not fully specified in integer arithmetic, but using floating-point operations (sinus and cosine samples, for the implementation of Discrete Cosine Transform - DCT) that introduce approximation errors due to the specific implementation. On one hand, the programmer can adapt and optimize the implementation on a particular architecture, but in the other hand the cost of this flexibility is the introduction of a prediction drift. To avoid this, H.264 standard introduces an integer transform in which all the operations are natively defined by fixed point arithmetic, thus without loss of information. Moreover, H.264 transform is applied to 4×4-pixel blocks, whereas the previous video coding standards used 8x8 blocks. This smaller block size leads to a significant reduction in ringing artefacts (image border noise) and computational requirements. In addition, compression gain is improved by using inter-block pixel prediction for intra-coded frames. The transform is applied to prediction residuals, reducing the spatial correlation and the size of transformed block without affecting the compression gain.

The length-4 transform proposed in H.264 is an integer orthogonal approximation of the Discrete Cosine Transform (DCT), which allows bit-exact implementation for both encoder and decoder. From a computational point of view, the new transform has the additional benefit of substituting multiplications with shifts, more suitable for the implementation on reconfigurable hardware. For improved compression efficiency, H.264 employs a hierarchical transform structure, in which the DC coefficients of neighboring 4x4 transform are grouped in 4x4 blocks and transformed again by a second level transform.

Integer Transform design

DCT is commonly used as block transform coding of images and video because its close approximation to the statistically optimal Karhunen-Loeve transform, for a wide class of signals. DCT maps a N-length vector x into a new vector X, by a linear transformation

$$X = Hx$$

where the elements of the matrix H are defined by

$$H_{kn} = H(k,n) = c_k \sqrt{\frac{2}{N}} \cos\left[\left(n+\frac{1}{2}\right)\frac{k\pi}{N}\right]$$

The DCT matrix is orthogonal, thus

$$x = H^{-1}X = H^T X$$

A disadvantage of DCT is that coefficients $H(k,n)$ are irrational numbers, that in a digital computer are approximated, thus introducing some degree of error. In H.264, the transform is based on the DCT and operates on $4x4$ blocks of residuals data, but differs from a DCT for the fact that is natively defined using integer arithmetic (without loss of accuracy, for both direct and inverse transform), avoiding mismatch between encoders and decoders. Furthermore, the core part is multiply-free, and a scaling multiplication is integrated in the quantizer thus reducing the total number of multiplications.

A $4x4$ DCT of an input array X is given by:

$$Y = AXA^T = \begin{pmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & -c \end{pmatrix} \begin{pmatrix} & & & \\ & X & & \\ & & & \\ & & & \end{pmatrix} \begin{pmatrix} a & b & a & c \\ a & c & -a & -b \\ a & -c & -a & b \\ a & -b & a & -c \end{pmatrix}$$

where:

$$a = \frac{1}{2}; b = \sqrt{\frac{1}{2}} cos\left(\frac{\pi}{8}\right); c = \sqrt{\frac{1}{2}} cos\left(\frac{3\pi}{8}\right)$$

The matrix multiplication can be factorized in the following form:

$$Y = (CXC^T) \otimes E =$$

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & d & -d & -1 \\ 1 & -1 & -1 & 1 \\ d & -1 & 1 & -d \end{pmatrix} \begin{pmatrix} & & & \\ & X & & \\ & & & \\ & & & \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & d \\ 1 & d & -1 & -1 \\ 1 & -d & -1 & 1 \\ 1 & -1 & 1 & -d \end{pmatrix} \otimes \begin{pmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{pmatrix}$$

CXC^T is the "core" 2D transform, while E matrix represents the required scaling factors ($\otimes$ means scalar multiplication) for the corresponding elements of CXC^T. a and b are the same defined before, while $d = \frac{c}{b}$ is approximatively 0.414. To simplify the implementation of the transform d is

approximated to 0.5, and b is consequently modified in order to maintain the matrix orthogonal, so that:

$$a = \frac{1}{2}; b = \sqrt{\frac{2}{5}}; d = \frac{1}{2}$$

Then, the 2nd and 4th rows of C and the 2nd and 4th columns of C^T are up-scaled by a factor of 2, post-scaling consequently the matrix E. The final forward transform becomes:

$$Y = (CXC^T) \otimes E =$$

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{pmatrix} \begin{pmatrix} \\ X \\ \\ \end{pmatrix} \begin{pmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{pmatrix} \otimes \begin{pmatrix} a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \\ a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \end{pmatrix}$$

Matrix E is collapsed in the quantizer, that is defined by:

$$Z_{i,j} = round(\frac{Y_{i,j}}{Qstep}) = round(W_{i,j}\frac{PF}{Qstep})$$

where PF is post-scaling factor that depends on the position (i, j) such that:

$$\begin{array}{ll} (0,0),(2,0),(0,2),(2,2) & a^2 \\ (1,1),(1,3),(3,1),(3,3) & b^4/4 \\ other & ab/2 \end{array}$$

On the decoder side, we can use H^T scaling the reconstructed transform coefficients in order to compensate the different row norms. On the other hand, we need to reduce the dynamic range gain in order to minimize the combined rounding errors from the inverse transform and reconstruction. H.264 standard scales the odd-symmetric basis functions by 1/2, replacing the rows $[2\ 1\ -1\ -2]$ and $[1\ -2\ 2\ -1]$ with $[1\ 1/2\ -1/2\ -1]$ and $[1/2\ -1\ 1\ -1/2]$, respectively. That way, the sum of absolute values of the odd functions is 3, which reduces the dynamic range gain for the 2-D inverse transform from 6^2 to 4^2. This allows reducing the dynamic range increase from 6 bits to 4 bits. Therefore, the inverse transform matrix is then defined as

$$\mathbf{H_{inv}} = \begin{pmatrix} 1 & 1 & 1 & 1/2 \\ 1 & 1/2 & -1 & -1 \\ 1 & -1/2 & -1 & 1 \\ 1 & -1 & 1 & -1/2 \end{pmatrix}$$

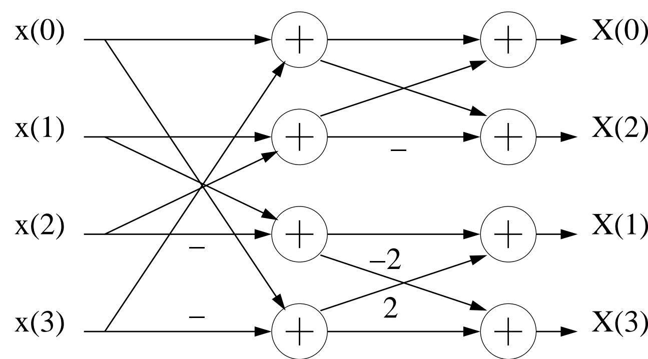
A key point is that the small errors caused by the right shifts are compensated by the 2-bit gain in the dynamic range of the input to the inverse transform. Inverse transform can thus be performed by the following expression:

$$x = H_{inv} X H_{inv}^T$$

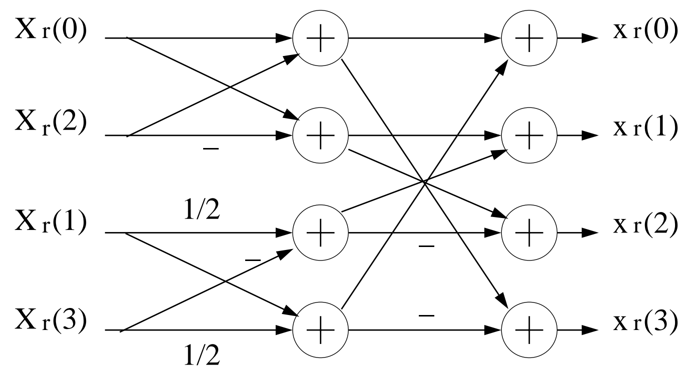
Direct transform mapping and optimization

Various methods has been proposed in literature in order to decrease the computational complexity of the (I)DCT, most of them based on decimation algorithms and *butterfly* structures. For the case of H.264 direct and inverse transform, the butterfly structure is represented by the schemes in Figure 6.22.

Direct transform is performed on the residual frame obtained from the pixel-to-pixel difference between the current frame and the previous frame, reconstructed by decoding the previously encoded frame. Therefore, before the butterfly structure is required an additional stage performing the pixel to pixel difference between corresponding macro-blocks. The following code represents the software implementation of the H.264 2-D 4x4-DCT.



Fast implemetation of direct transform (a)



Fast implemetation of inverse transform (b)

Figure 6.22: Fast implementation of the 1-D H.264 transform

```
for( y = 0; y < 4; y++ )
    for( x = 0; x ¡ 4; x++ )
        d[y][x] = pix1[y][x] - pix2[y][x];

for( i = 0; i < 4; i++ ){
    int s03 = d[i][0] + d[i][3]; int s12 = d[i][1] + d[i][2];
    int d03 = d[i][0] - d[i][3]; int d12 = d[i][1] - d[i][2];
    tmp[0][i] = s03 + s12; tmp[1][i] = 2*d03 + d12;
    tmp[2][i] = s03 - s12; tmp[3][i] = d03 - 2*d12;
}

for( i = 0; i < 4; i++ ){
    int s03 = tmp[i][0] + tmp[i][3]; int s12 = tmp[i][1] + tmp[i][2];
    int d03 = tmp[i][0] - tmp[i][3]; int d12 = tmp[i][1] - tmp[i][2];
    dct[0][i] = s03 + s12; dct[1][i] = 2*d03 + d12;
    dct[2][i] = s03 - s12; dct[3][i] = d03 - 2*d12;
}
```

¿From a computational point of view, the direct transform requires pixel-to-pixel subtractions (highly parallel), sums and shifts. Multiplications are not required since the matrix coefficients can be strength-reduced in shifts and sums. Two-dimensional DCT can be performed using the common row-column algorithm, thus the whole computation schema is that one represented in Figure 6.23, where the B_i block are the previously described 4-point 1-D DCT.

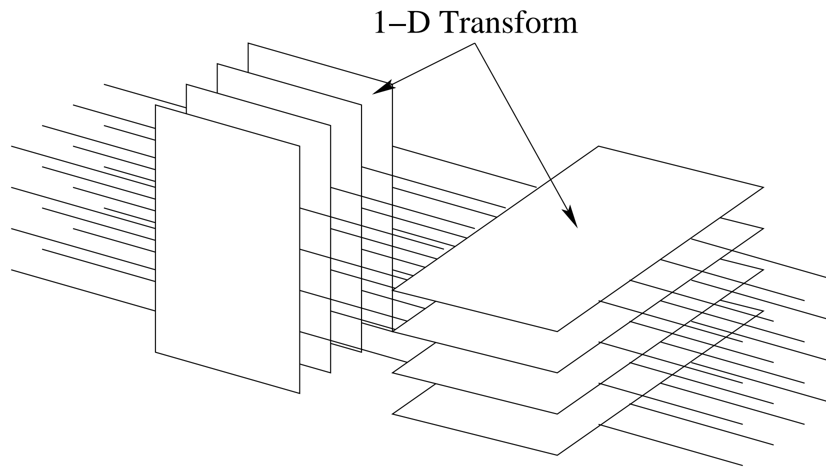


Figure 6.23: Fully-unfolded bi-dimensional transform diagram

Considering the data range, the elements of the transformed matrix can be represented by 15 bit, since 9 bits are required after the pixel-to-pixel difference (pixel are represented by 8-bit variable) and 6 additional bits are required after the transform. Therefore the representation of the whole matrix requires a bandwidth greater than the bandwidth available in PiCoGA-III, that provides up to 4 32-bit outputs. This requires to split the execution of the transform in two successive calls, each of them providing one half matrix. To save memory space and to improve the communication bandwidth, two pixels can be packed in the same output word without loss in precision. Therefore, the implementation on PiCoGA-III requires the insertion of a multiplexing layer for the selection of the outputs. The most efficient solution is to insert the multiplexing layer at the same level of the row-column transposition, as is shown in Figure 6.24:

- the row computation is performed by a full-unfolded schema, since

for each column transform a sample of every row is required.

- the column computation is performed by a 2-way unfold, since this level of unfolding is the maximum allowed by the output bandwidth. The multiplexing layer allows to choose the couple of rows under elaboration.

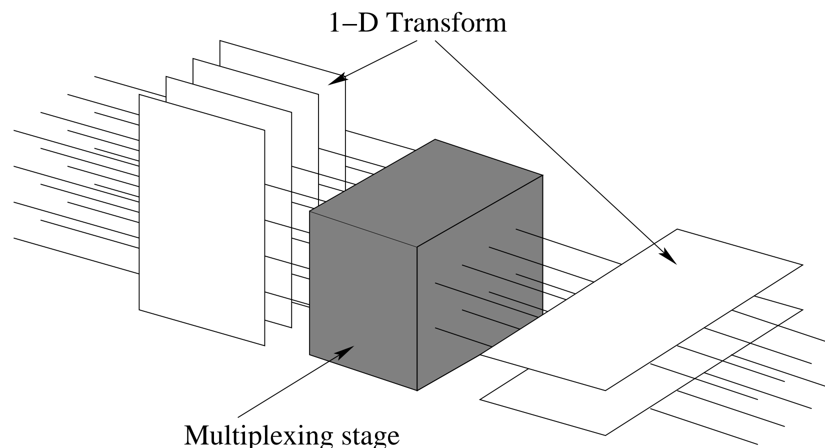


Figure 6.24: Partially unfolded 4x4 DCT schema

This function features 9 inputs (4+4 for the two input blocks with 8-bit pixels packed in 32-bit words, 1 for the multiplexing) and 4 outputs. To complete the elaboration of one 4x4 block is required to call two times this function. The static features are summarized in the table 6.7, while Figure 6.25 shows the mapping on the array. A detailed analysis of this implementation emphasizes resources under-utilization in the reconfigurable device. As can be seen in Figure 6.25, some rows is only partially used, as for example the case of the 3,7 and 11.

Although not critical in term of performance, the under-utilization of the device can be seen as an overhead in term of area and can cause an additional energy consumption, which is roughly proportional to the number of active rows. It is possible to use a software pipelining methodology in order to fold cascaded pipeline stages in the same pipeline stage, by means of the introduction of status register and feedbacks which allow to work with data referred to a different iteration time. Software pipelining

Pgaop name	sub4x4dct
Rows	20
Pipeline Stage	7
Latency	8
Issue Delay	1

Table 6.7: sub4x4dct

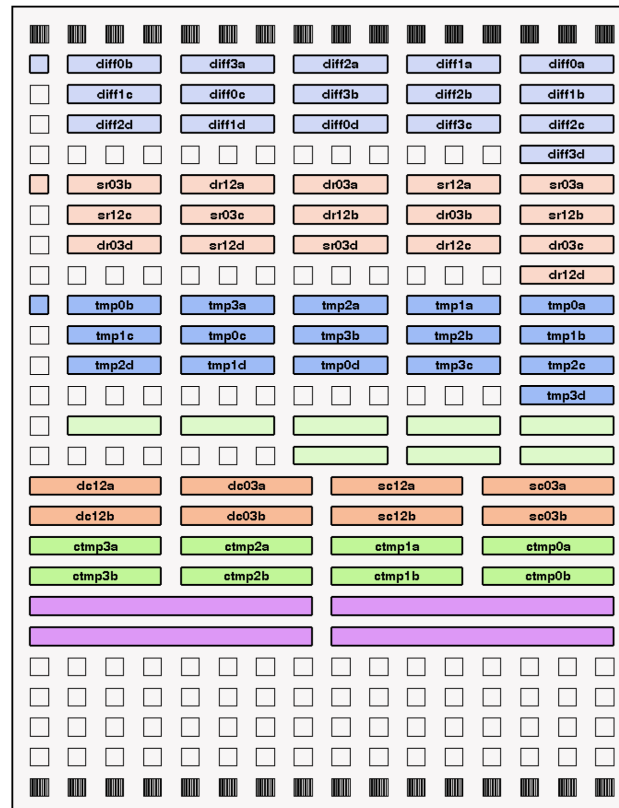


Figure 6.25: sub4x4dct rows occupation

the first two stages, and the third and fourth ones, we can save two rows, as it is shown in Figure 6.26. As a consequence, the PiCoGA operation shall be call more times in order to both fill the internal status register with valid data and to output the results. Since this process is pipelined, it introduces only a small overhead due to this prologue/epilogue requirements. After the first three additional calls, this operation provides as output a half matrix every cycle. The static features of this implementation

are summarized in the table 6.8.

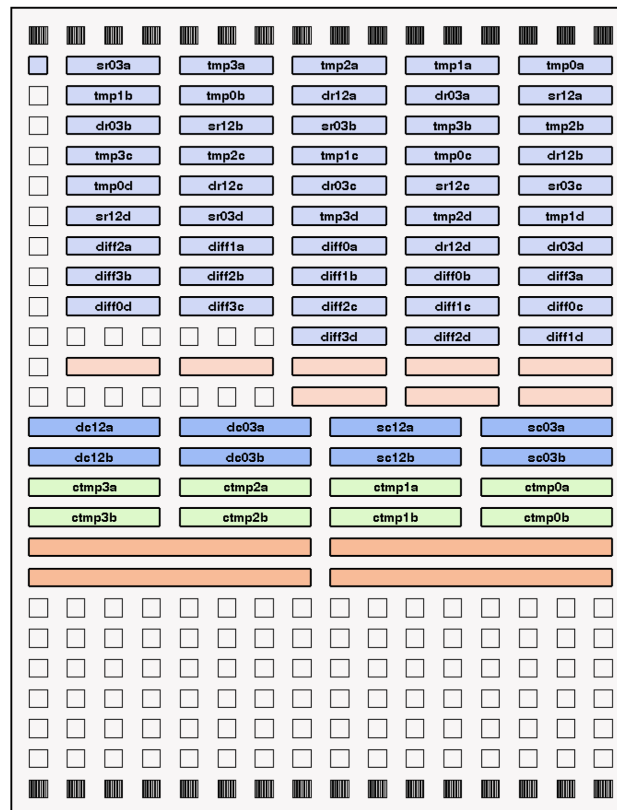


Figure 6.26: Modified sub4x4dct for area optimization

Pgaop name	**sub4x4dct**
Rows	18
Pipeline Stage	5
Latency	6
Issue Delay	1

Table 6.8: sub4x4dct

Inverse transform mapping and optimization

As for the direct transform, the inverse transform is split in two computational kernels: in the first part, given the transformed matrix, is extracted

the residual matrix, while in the second part the reconstructed block is added. Since the input data have a range that is greater than the range of the direct transform, area requirements are more demanding and the optimized implementation requires two PiCoGA operations. The basic butterfly schema is that one shown in Figure 6.22(b). Furthermore, a further stadium shall be added just before the output, in order to perform a shifting and rounding operations, as represented in Figure 6.27.

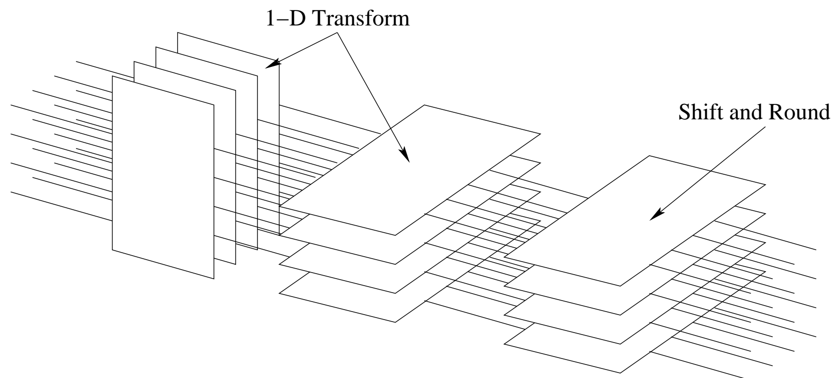


Figure 6.27: Fully-unfolded inverse 4x4-IDCT basic diagram

Also in this function the output data range not allows to map the whole function on PiCoGA-III, since the matrix elements require 13 bit to be represented (9 for the difference representation and 4 due to transform and inverse transform). The methodology adopted to solve this problem is the same applied to the previous PiCoGA operation, with the introduction of a multiplexing layer to select the required outputs after each PiCoGA operation elaboration, as in Figure 6.28.

The last stage of this function performs a shift-and-round operation. In the software implementation this operation is obtained adding 32 (0b10000 in binary) and right-shifting by 6 bits. In the hardware implementation, in order to reduce the area occupation it is possible to carry out part of the operation of right-shifting before the sum, thus reducing the number of bits required. After the shift-and-round, a clipping operation is performed, setting to 0 every negative value, and to 255 every value greater than 255. Let us suppose to have in input 16 bit data, the clipping operation can be performed by the logical structure represented in Figure 6.29.

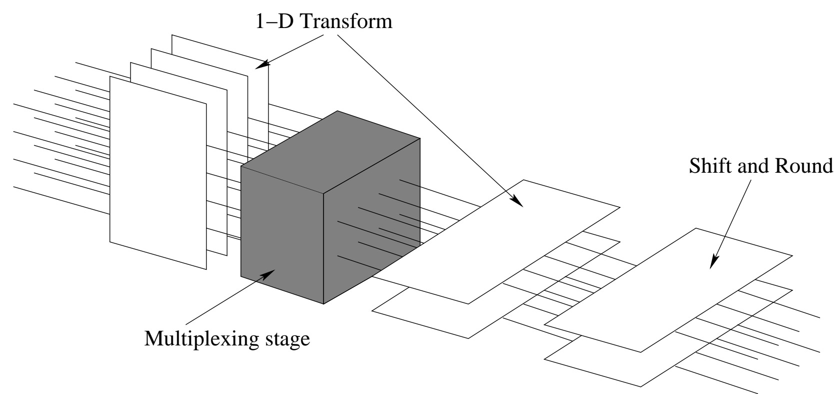


Figure 6.28: Partially-unfolded inverse 4x4-IDCT basic diagram

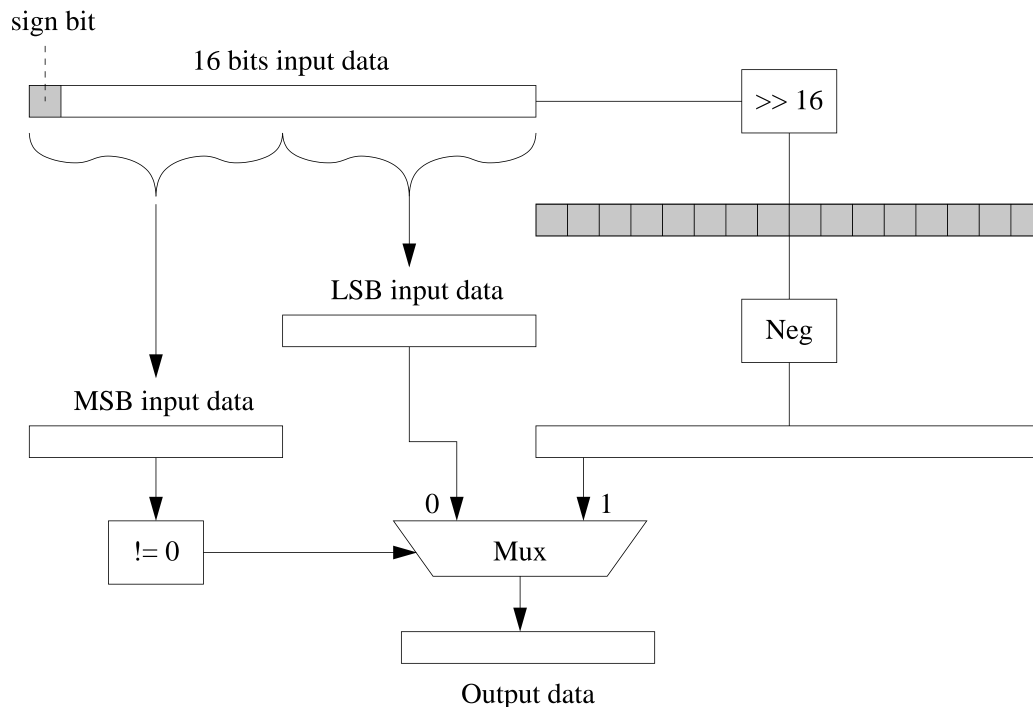


Figure 6.29: Modified clipping function structure

In this structure, the multiplexer selector is determined analyzing the 8 most significant bits of the input data. If these bits are set to 0 the clipping shall not be performed (the number is positive and less than 255), then the output is obtained passing the input data through the clipping structure. Otherwise the clipping shall be performed, and the output data are obtained by 16-bit shifting and not operations. The first operation generate

a binary number composed from all 0 if the input data is positive, or all 1 otherwise. Therefore, the not operation allows to obtained in output all 1 (corresponding to 255 if we consider such number unsigned char) if the input data is positive and all 0 (corresponding to 0) otherwise. This type of implementation allows to use only 4 rows of PiCoGA-III for the computation of 16 value clipping. The high issue delay requires the insertion of some retiming registers that caused a increase of the used rows (from 4 to 8).

Summarizing, two functions are used to implement the inverse 4x4 DCT. The first one is characterized by 8 inputs, containing the input matrix composed by 16 elements of 16 bits, and 4 outputs returning the two selected columns. The second one is characterized from 8 inputs, 4 for each input matrix (residual and reference), and 4 outputs containing the output block composed from 16 pixels of 8 bits. To elaborate a 4x4 block inverse DCT is necessary to call two times the first function and only one the second one. The static features of the two PiCoGA operation are summarized in the table 6.9.

Pgaop name	**F4x4idct**
Rows	22
Pipeline Stage	7
Latency	8
Issue Delay	1
Pgaop name	**add4x4**
Rows	14
Pipeline Stage	4
Latency	5
Issue Delay	1

Table 6.9: F4x4idct and add4x4

Results

Both direct and inverse transform are implemented in such a way that is suitable for the computation in a pipelined form. In the case of the direct transform, only one PiCoGA operation is required. Therefore, it is possible to feed the reconfigurable device with new data every cycle, by mean of the high-bandwidth direct memory access performed via programmable address generators available in the DREAM adaptive DSP. A set of blocks is stored in the local memory, then the computation start. Depending on the number of locally stored and elaborated macro-blocks, also defined as interleaving factor, the computation achieves better performance figures since the pipelining is better exploited thus allowing a reduction of stalls.

On the contrary, in the case of the inverse transform, two PiCoGA operation are required. To pipeline as much as possible the computation an intermediate data repository is necessary. The local data buffers of DREAM can be used for this purpose, thus running the two PiCoGA operations alternatively and storing/reading intermediate data from the exchange buffer.

Fig. 6.30 shows the performance improvement (speed-up figure) with respect to a RISC processor working at the same frequency. As expected, the speed-up increases with the interleaving factor, saturating when the PiCoGA pipeline is completely active and prologue/epilogue overhead are negligible compared to the overall computation time.

Fig. 6.31 shows the throughput achieved with respect to the interleaving factor. Since the direct transform provide one half output matrix for every PiCoGA trigger, the maximum bandwidth achievable is:

$$B_{max} = \frac{2calls * 4UsedOutputs * 32bit * 200MHz}{2calls} = 25.6Gbit/sec$$

In the case of the inverse transform, two PiCoGA operations are required thus the maximum achievable bandwidth is 12.8 Gbit/sec. As it is shown in Fig. 6.31, sizing properly the interleaving factor, near-optimal performance is achieved. The exploitation of the pipelining degree cause an increase of the dynamic power that is roughly proportional to the number of active rows per cycle and the memory bandwidth utilized. On the

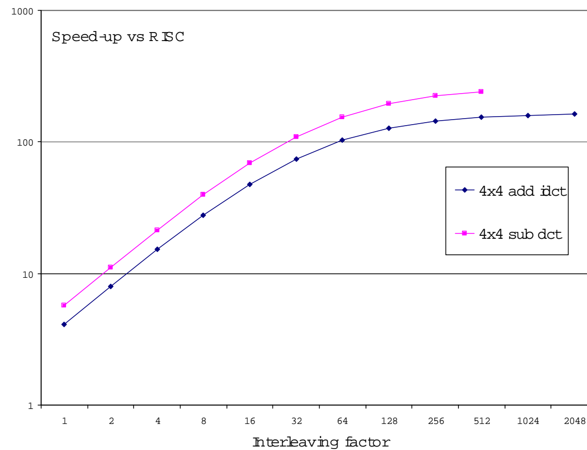


Figure 6.30: Speed-up figure with respect to a RISC processor working at the same frequency

contrary, the energy efficiency, mixing the energy consumption with the achieved performance, increase with the interleaving factor since the performance gain is greater than the energy increase. The related figure of merit is reported in Fig. 6.32.

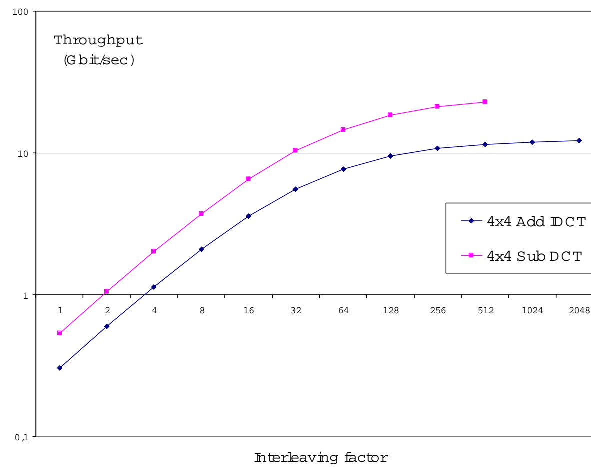


Figure 6.31: Throughput achieved with respect to interleaving factor

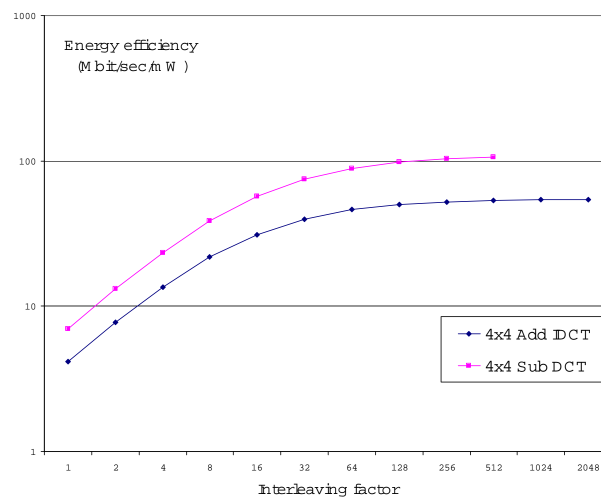


Figure 6.32: Energy efficiency with respect to interleaving factor

6.2.4 H.264 intra prediction with Hadamard transform for 4x4 blocks

Intra-frame prediction is introduced in H.264 advanced video coding standard in order estimate 4x4 pixel block starting from the neighboring pixels, as shown in Fig. 6.33. If compared to previous standard, as the JPEG2000, this enhancement allows to achieved better compression gain and the same signal to noise ratio, as proof in [126]. Although different block sizes are supported by the standard, for the base profile the commonly used macroblock is 4x4 pixels, and the intra prediction is applied to the luminance component (or luma), that represents the grey-scale.

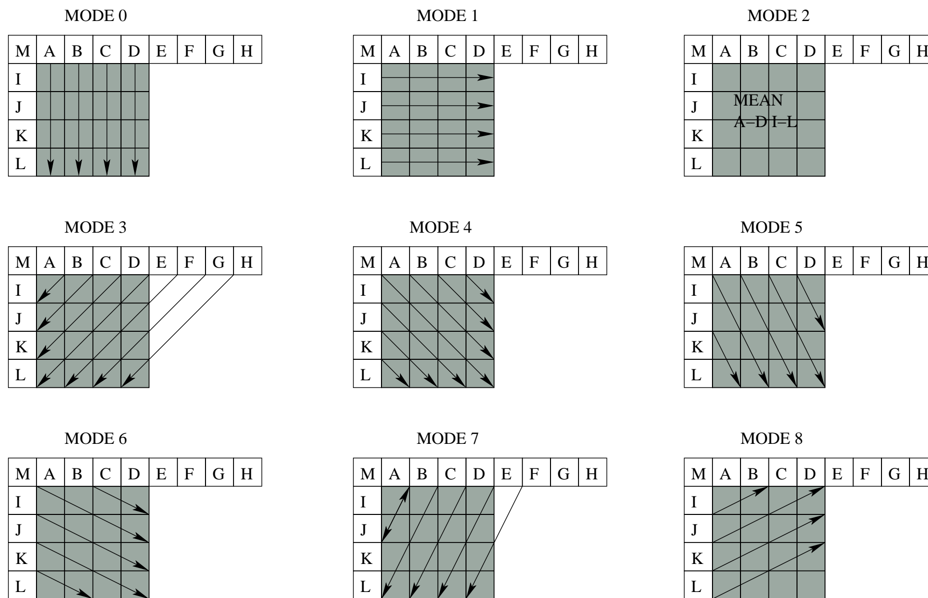


Figure 6.33: Intra prediction modes for 4x4 luma block

Although H.264 not specifies any mode decision, some algorithms are proposed with different trade-off between quality and computational complexity, considering both distortion and rate. While high complexity mode a sum of square differences is used, for low complexity mode decision, distortion is evaluated by sum of absolute differences (SAD) or sum of absolute transformed differences (SATD) between the predictors and original pixels (where the applied transform is the Hadamard transform). Usually, the coding performance by selecting SATD is 0.2 0.5 dB better. The rate is

estimated by the number of bits required to code the mode information, and depends - among the others - to the level of quantization. Most of the computational complexity is associated to SAD and SATD elaboration, and these two kernels are considered for the implemented on DREAM.

4x4 SAD mapping and optimization

Similarly to MPEG-2 standard, the SAD function is defined by:

$$SAD = \sum_{i=1}^{4}\sum_{j=1}^{4} |a_{i,j} - b_{i,j}|$$

where $a_{i,j}$ and $b_{i,j}$ are the $(i,j)^{th}$ elements of two macro-blocks A and B. SAD features a very high level of intrinsic parallelism, both at instruction level and at loop level. Instruction level parallelism resides in the arithmetic properties of the mathematical definition, whereas the loop level parallelism is due to the fact that SAD computation could be applied in parallel to all the macro-block in a frame. Each pixel is represented by 8-bit variable, thus it is possible to pack in 4 32-bit words a whole macro-block. Since PiCoGA-III features up to 12 32-bit input words, it is possible to transfer every cycle all the data required for the 4x4 SAD computation. The output is the SAD value, that can range between 0 and $16 * (256 - 0) = 4096$. Figure 6.34 shows the simplified (fully-unfolded) block diagram of the 4x4 SAD subdivided in the three phases required: differences, absolute value computation and adder tree.

This operation fits the PiCoGA-III computational capabilities, and achieved static features are summarized in table 6.10.

Pgaop name	**sad4x4**
Rows	10
Pipeline Stage	6
Latency	7
Issue Delay	1

Table 6.10: 4x4 Sum of Absolute Differences (SAD)

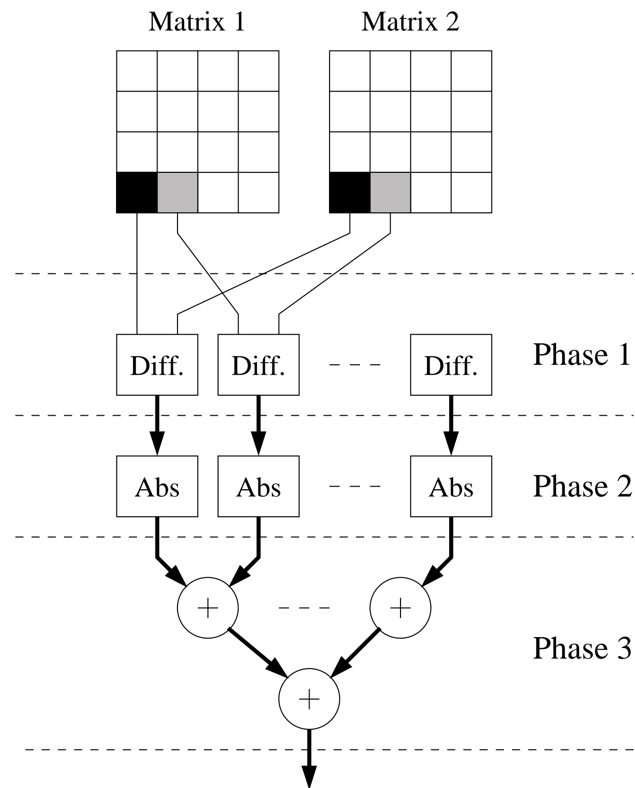


Figure 6.34: PiCoGA SAD structure

SATD mapping and optimization

The Sum of Absolute Transformed Differences (SATD) is defined as:

$$SATD = \sum_{i=1}^{4} \sum_{j=1}^{4} |c_{i,j}|$$

where $c_{i,j}$ denotes the $(i,j)^{th}$ elements of the C matrix, which is the Hadamard-transformed difference between the macro-block A and B. Considering the case of 4x4 pixel blocks, the Hadamard transform of the D matrix is calculate by:

$$C = H_4 D H_4^T$$

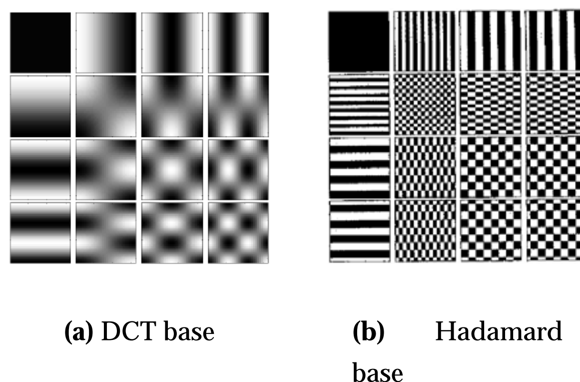
where H_4 (Hadamard transform) is defined by the orthogonal matrix:

$$\mathbf{H} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}$$

Since H_N has N orthogonal rows $H_N H_N^T = I_N$ (where I_N is the NxN identity matrix) and $H_N^{-1} = H_N/N$. Therefore, the inverse transform is defined by

$$D = H_4 C H_4^T$$

Investigating the number of sign transition among the values of each column of H_4, it is possible to see that the first one has 0 sign transitions, the second one 3 sign transitions, the third one 1 sign transition, and the fourth one 2 sign transition. The number of sign transition is often termed "sequence", and it is a common concept already present in Fourier transform (see Fig. 6.35). Zero sign transition corresponds to a DC component, whereas a big number of sign changes corresponds to high frequency components.



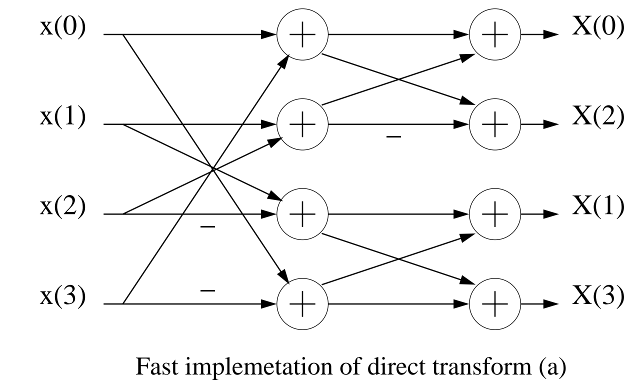
(a) DCT base **(b)** Hadamard base

Figure 6.35: DCT and Hadamard transform

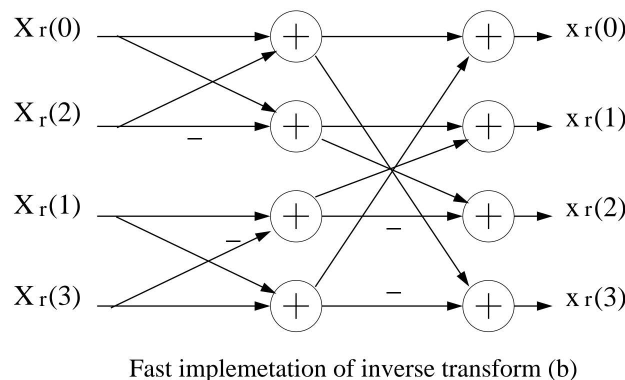
If the columns of H_4 are arranged per increasing sequence, the obtained matrix is called *Walsh transform matrix* and is defined as:

$$
\mathbf{W} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix}
$$

Just like in 4x4 DCT, the resulting basic computational engine has the usual *butterfly* structure, and the bi-dimensional transform is obtained by a row/column algorithm. The butterfly structure for the 1-D Hadamard transform is depicted in Fig. 6.36, where both direct and inverse data flow graph are represented.



Fast implemetation of direct transform (a)



Fast implemetation of inverse transform (b)

Figure 6.36: 1-D Hadamard transform butterfly schema

The 4x4 SATD operation is very similar to the previously described 4x4 SAD function, with the difference that the SATD performs a transform

before the computation of the absolute values. Fig. 6.37 shows the corresponding data flow graph.

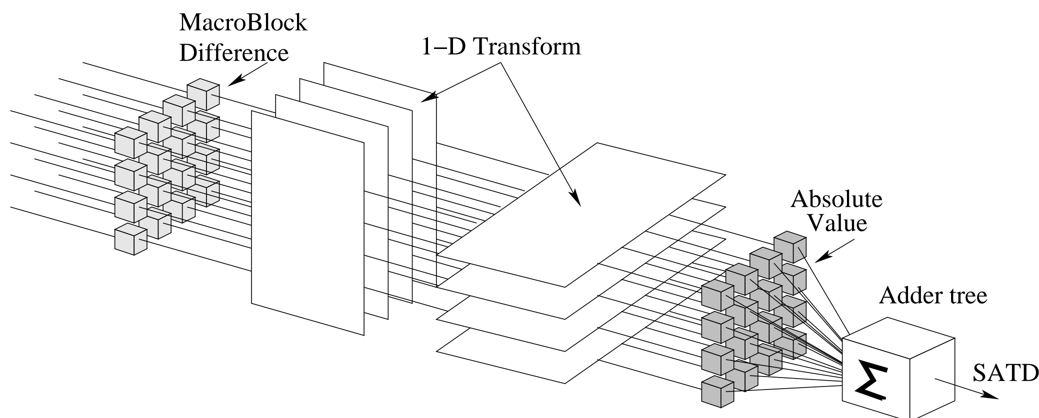


Figure 6.37: Fully unfolded 4x4 SATD data flow graph

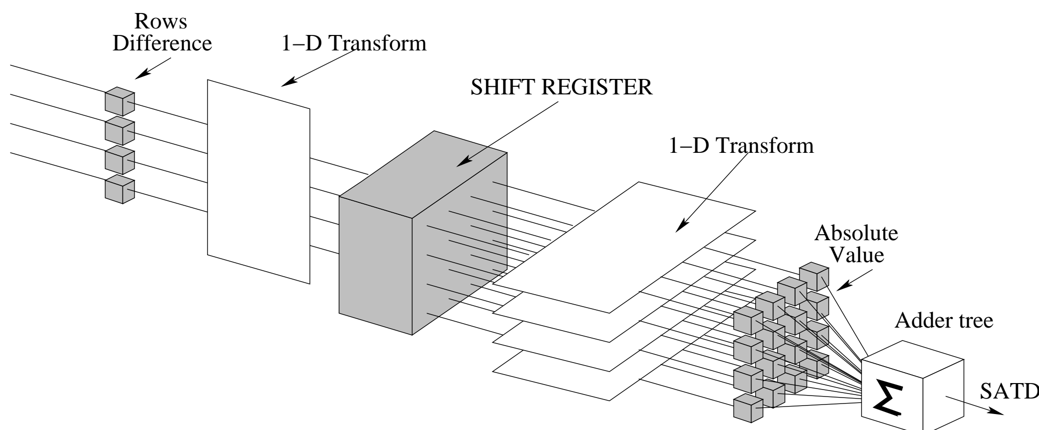


Figure 6.38: Partially folded 4x4 SATD block diagram

Although no limitation is given by I/O resources, the mapping of a whole 4x4 SATD does not fit the resource available on PiCoGA-III, then is required some kind of partitioning or folding. In particular, we chosen to fold the butterfly structure allowing to elaborate only one row per call, and to maintain unfolded the absolute value computation and the adder tree. In the middle, a shift register store the intermediate results, allowing to write-back a SATD computation every 4 cycle. Fig. 6.38 shows the block diagram of this implementation, while the required shift register is depicted in Fig. 6.39.

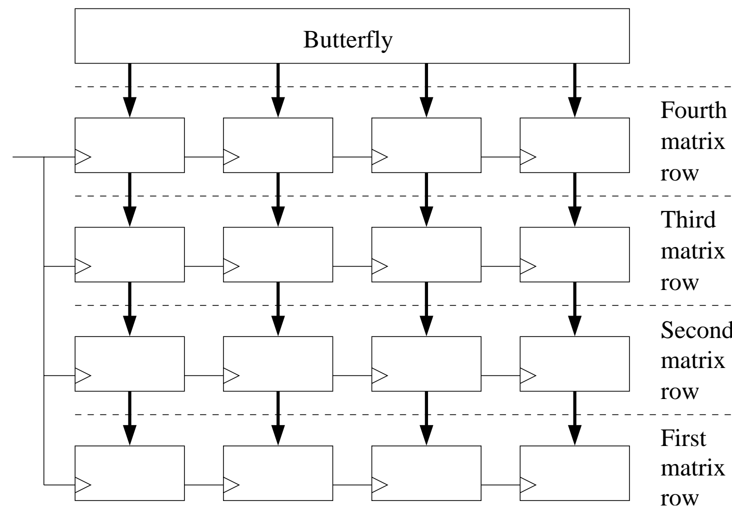


Figure 6.39: Shifter register structure used for the matrix transposition

Standing this computational structure, four calls are necessary to obtain a valid SATD computation since only after this time the internal shift register, implementing the transposition matrix, is filled with valid data. This notwithstanding, a streaming work-plan is allowed, although requiring output sub-sampling. Furthermore, software pipelining and retiming registers are used to improve the utilization of PiCoGA-III. Fig. 6.40 shows the mapped operation, while Table 6.10 summarizes the static performance.

Pgaop name	**satd4x4**
Rows	24
Pipeline Stage	6
Latency	7
Issue Delay	1

Table 6.11: 4x4 SATD static performance

Results

Results achieved implement 4x4 SAD and SATD function on the DREAM architecture are reported in Fig. 6.41, Fig. 6.42 and Fig. 6.43. In particular,

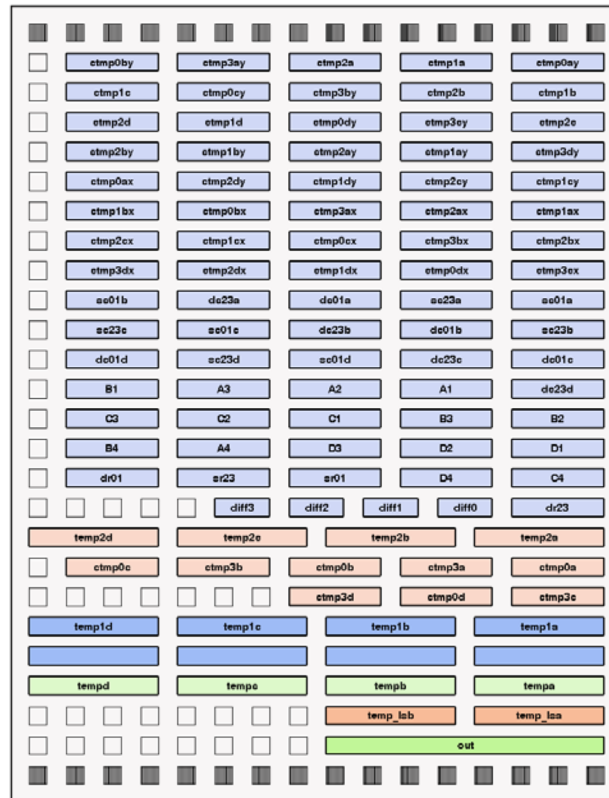


Figure 6.40: Optimized SATD mapping

Fig. 6.41 shows the speed-up figure with respect to a RISC processor working at the same frequency of DREAM, while Fig. 6.42 reports the absolute throughput achieved. Fig. 6.43 shows the energy efficiency, measured in term of Mbit/sec/mW. The last figure of merit is the inverse of nJ/bit, value that gives an idea of the amount of energy spent for the elaboration of each bit in output.

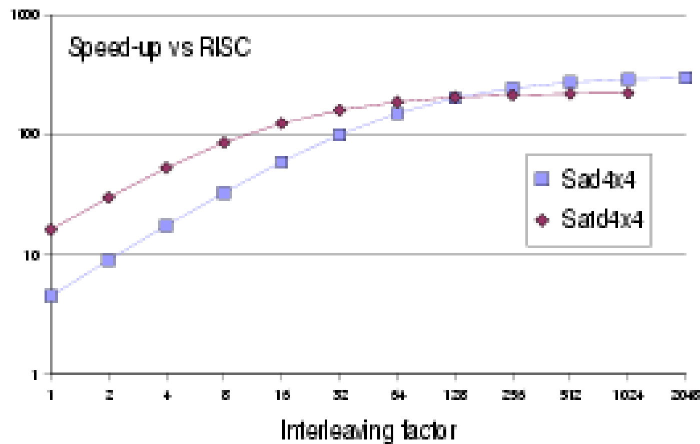


Figure 6.41: 4x4 SAD and SATD speed-up figures with respect to the interleaving factor

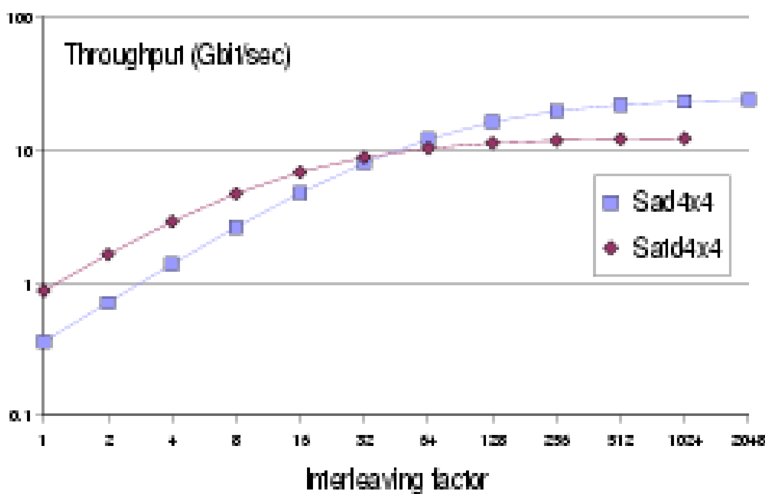


Figure 6.42: 4x4 SAD and SATD throughput with respect to the interleaving factor

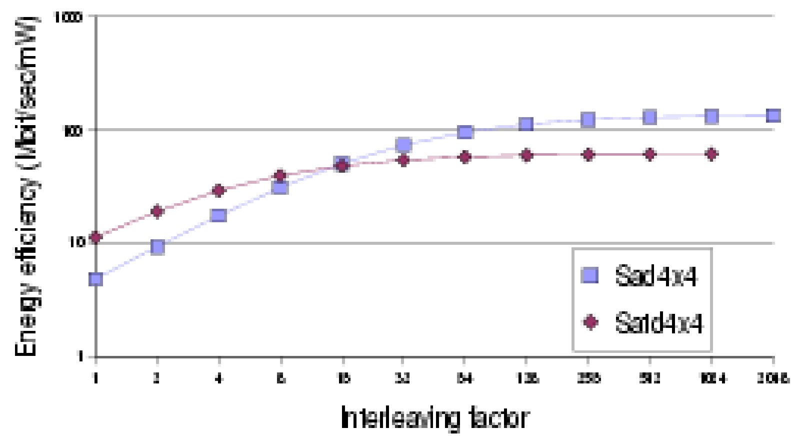


Figure 6.43: 4x4 SAD and SATD energy efficiency with respect to the interleaving factor

Chapter 7

Performance and development time trade-offs

Application development on reconfigurable processors is performed partitioning the computational workload between software and (reconfigurable) hardware. As usual, partitioning is an iterative process of design refinements or (in the worst-case) re-designs in order to achieved the optimal result. But, what is the optimal result? Of course, we can said that constraints, such as real-time requirements, shall be verified, but it could be possible to further improve the performance in order to achieve greater energy reduction. As an example, additional performance improvements can be used to over-boost the application by reducing the pure cycle count, then reducing the working frequency to achieve better energy consumption figures. In the case in which complex applications feature a set of computational kernels (and not only one critical hot spot), it is required to speed-up as much as possible all the kernels or it is better to focus the optimization in a subset of them?

Of course, performance, be it computation speed or energy consumption, is not the only cost function to be evaluated, if performance improvements are generated by additional development, thus additional development costs. As for the well-known assembly-level optimization, the programmer shall choose for each critical kernel the most appropriate development methodology. As shown in the previous chapter, the effective

utilization of reconfigurable devices could be driven by a software or a hardware approach that depends on the application field can return different performance.

The quantitative analysis of the different performance and development time trade-off has been evaluated on the XiRisc reconfigurable processor. The adopted functional unit computational model reduces any communication overhead between processor core and reconfigurable device. This allows to obtain interesting performance improvement after few hours or few days of work, by mean of local optimizations obtained by re-mapping part of the software code on the reconfigurable device. On the contrary, hardware approaches, deeply investigating the mathematical aspects of the computation to match the device capabilities, promise impressive performance improvements at the cost of long development time. Commonly, a hardware approach could imply additional skills with respect to backgrounds on software development and this can be seen as an additional cost.

Several applications were developed in order to evaluate the different trade-off points proposed by the Griffy methodology in terms of performance, required skills and development time. While previous works, such as [73, 74], detailed the implementation of common applications on the XiRisc reconfigurable processor, and [68, 66, 67] summarize performance and energy reduction on typical benchmarks compared to traditional general-purpose embedded processors and DSPs. In this section, we discuss the relation between the performance achieved and the development time spent working on the XiRisc.

The applications were chosen in order to be representative of the whole “embedded scenario”, using open-source codes that are part of well-known benchmark suites such as MediaBench [57] or from well-known applications in the field of image processing, telecommunications, and cryptography. Each algorithm has been optimized for XiRisc architecture, exploiting reconfigurability as much as possible. In this case, Griffy-C code is used as entry-point to configure the PiCoGA, refining at low-level the initial configuration using if necessary LUTs or built-in functions.

Algorithm	Speed-up after development time					Line of Code		Methods of Developments
	$\frac{1}{2}$ day	1 day	10 days	>1 months	>3 months	SW Only	Griffy-C	
IDEA	1,9	2,1	2,6	2,7	-	600	740	Clustering, Mult
CRC	2,3	2,6	2,6	4,0	-	154	430	Clustering, LUT
RSA	1	1,1	1,3	1,6	-	2500	130	Clustering, Pipelining
AES (128-bit)	1	1	1,5	2,5	-	860	490	Clustering, LUT
Kasumi	1,1	1,5	1,6	2,4	-	507	391	Clustering, LUT
Reed-Solomon Encoder (255,239)	3	4	7	10	80	156	1500	Clustering, LUT, HW-approach
Viterbi Decoder	1,4	1,6	1,7	3	-	300	250	Clustering, LUT
FDCT	1,5	1,6	1,8	2	2,5	250	1000	Clustering, LUT
IDCT	1,2	1,5	2	2,5	-	260	250	Clustering
Quantization	2	2,5	-	-	-	400	1000	Clustering, Pipelining, Mult
VLC	1,5	2,1	-	-	-	400	24	Clustering, Pipelining
Motion Estimation	1,5	3	7	14	16	350	300	Clustering, Loop unrolling, Pipelining, Loop transformation
MPEG-2 Encoder	1,2	1,5	2	2,5	5	10000	2800	-
MPEG-2 Decoder	1,2	1,3	1,4	1,5	-	4000	120	-
Find_Best	1,5	2,5	5,4	-	-	200	50	Clustering, Pipelining
Find_Acbk	1,1	2,2	4,5	-	-	200	50	Clustering, Pipelining
Estim_Pitch	1,3	2,3	4,8	-	-	200	50	Clustering, Pipelining
Vocoder G.723.1	1,1	1,2	1,9	2	-	16500	210	-
Residu	1,5	3,5	5,0	-	-	83	144	Clustering, Loop unrolling
Vocoder ETSI GSM 06.60	1,1	1,5	1,8	2,6	-	15000	169	-
Template Matching Ncc	1,1	1,5	4	4,4	-	1700	291	Clustering, Pipelining
Template Matching Bpc	1,1	1,2	1,5	1,7	-	1900	291	Clustering, Pipelining
Average	1,5	2	3	3,8	7,5	2690	500	-

Table 7.1: Experimental results on application development

Reported results include both whole applications and many significant critical kernels. In all cases, most of the work was performed by undergraduate students with about 1-2 weeks of training in reconfigurable computing, and the development was stopped when an implementation providing near-optimal results was obtained. This was either because all the computational resources were fully exploited, and hence no further performance improvement could be achieved with the same architecture, or because further improvements would have been too expensive in terms of development time.

Table 7.1 shows the performance improvement in terms of speed-ups compared to a VLIW RISC processor, featuring the same instruction set as XiRisc but not augmented by PiCoGA. In this case, the speed-up takes into account only the pure cycle count, considering core-only and PiCoGA-augmented processors as working at the same clock frequency. The table also reports the number of additional Griffy-C code lines required for the final implementation compared to the amount of initial code. This should give an idea of the amount of work required to achieve the final solution. Table 7.1 also indicates the main methodology and implementation techniques providing the most significant performance gain for the specific application.

The same results are summarized in Figure 7.1. Two zones in the graph can be clearly distinguished, one corresponding to pure software optimizations, such as loop unrolling and memory reorganization, and one corresponding to more hardware-oriented optimizations. In less than 2 days of work, one can easily obtain 2-3$\times$ speed-ups using the automated C-to-DFG translation and standard techniques (software pipelining and loop unrolling). Longer development times, however, leads to much larger performance improvements, up to one order of magnitude. The highest speed-ups are achieved through manual optimization, rescheduling assembly instruction, careful algorithm analysis, and, in extreme cases, even hardware synthesis.

For some applications, the maximum performance gain is obtained by re-writing the algorithm from scratch, following the analysis of its mathe-

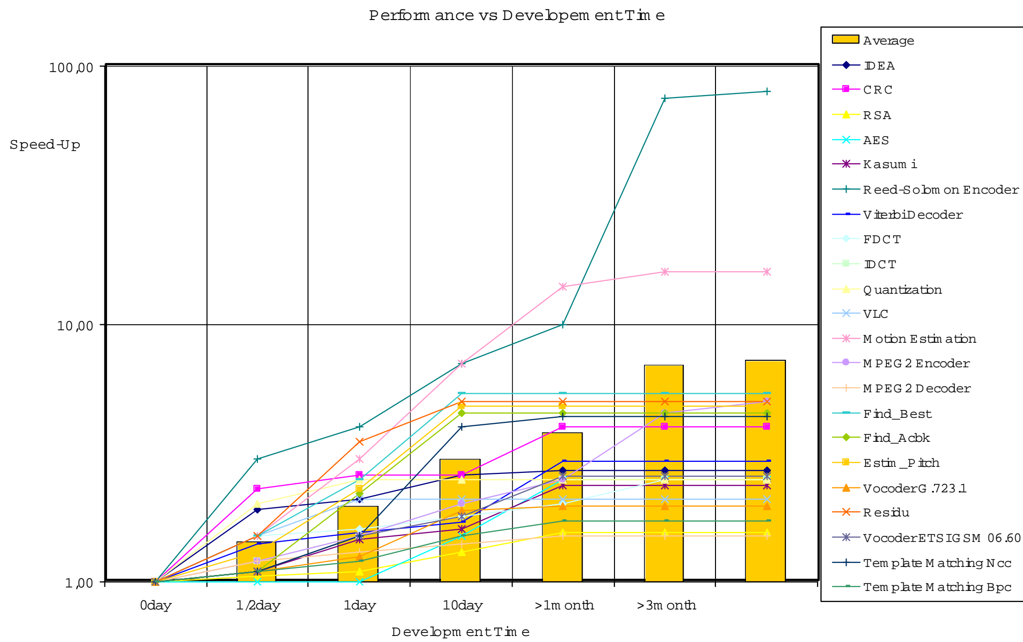


Figure 7.1: Application development trade-off

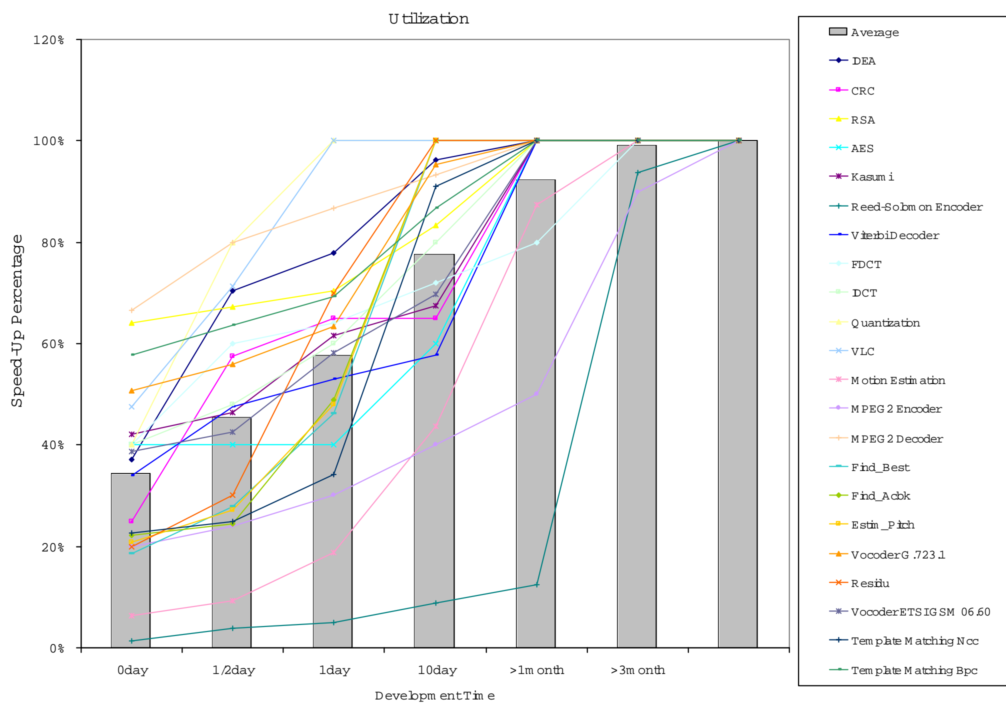


Figure 7.2: Development Time vs Speed-Up percentage

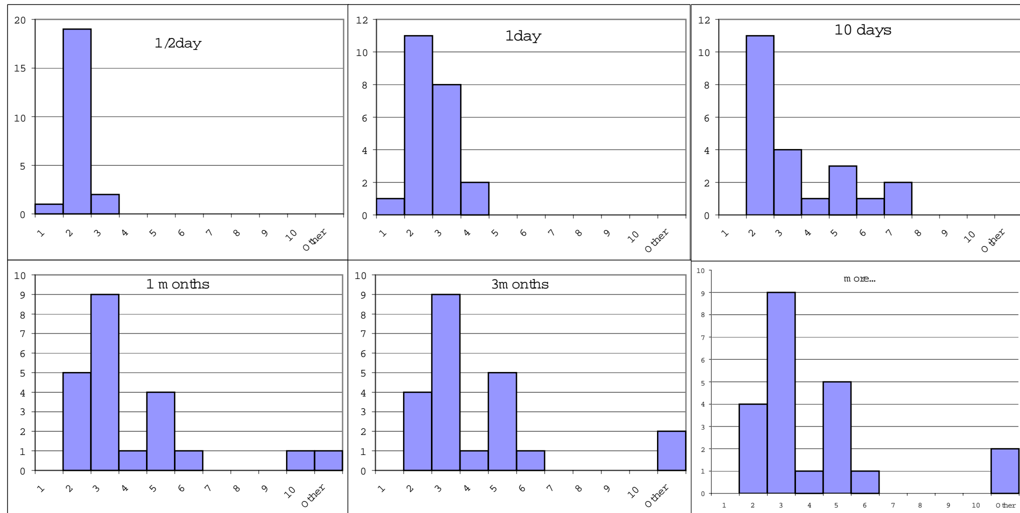


Figure 7.3: Distribution of speed-up with respect to development time

matical structure. This is, for example, the case of the Reed-Solomon Encoder, where software implementations are performed using 256-element hash-tables and arithmetical operations, while a hardware approach requires one to directly implement operations in Galois Fields arithmetic using LUTs. The development time reported includes both the time required to develop the new algorithm and the time required to implement and validate it.

Figure 7.2 shows the percent average speed-up obtained by spending additional time, in term of the final implementation. About 70% performance gain is obtained in less than 10 days. Figures 7.2and 7.1 can guide a designer to estimate a cost-effective development strategy and an appropriate trade-off between expected performance and required development time and expertise, balancing for example the time spent for each kernel in a whole application.

Figure 7.3 shows the distribution of performance speed-ups with respect to the development time spent. While the average speed-up can be affected by the performance of biased-applications, the distribution of speed-ups indicates the number of algorithms that achieve a performance improvement in a specific range. In the short-term, the distribution is concentrated around 2-3× speed-up figures, while in long-term development

Algorithm	Speed-Up	Energy Saving	Development Time on DSP
MPEG-2 Encoder	1.9×	63.3%	11h
MPEG-2 Decoder	1.2×	58.4%	3h
IDCT	1.9×	73.8%	1h
IDCT(1)	0.8×	-160%	-
Motion Estimation	2.3×	64.8%	7h
Reed-Solomon Encoder (255,239)	80×	80.1%	20h
Residu(2)	0.2×	-340%	15h
AES	1.02×	20%	20h
IDEA	0.85×	0.11%	1h
RSA(3)	1.2×	69.8%	3h

(1) Using the 16-bit TI C5510 and the assembly code provided by the optimized DSP library
(2) Using the dedicated application-specific instruction set
(3) 1024 bit key and 2KByte message

Table 7.2: XiRisc vs. TI TMS320C6713 Performance Comparison

three main regions can be identified, depending on the algorithm features. The first peak in the long-term histogram is typical of algorithms having little parallelism or not well-suited to PiCoGA, while applications with a high degree of instruction- and data-level parallelism benefit by more than 1 month of development time (achieving an average performance peak of 5×). Algorithms that require one to re-design the application adopting a hardware-oriented approach can achieve speed-ups greater than 10× (3rd peak in the histogram) but the development time usually takes more than 3 months.

Table 7.2 shows a performance comparison between the XiRisc reconfigurable processor and a TI TMS320C6713 32-bit general-purpose DSP capable of executing up to 8 32-bit instructions per cycle [96]. It can be observed that in many cases XiRisc achieves a better performance in terms both of speed and energy. On the other hand, as expected, the DSP performs better in heavily MAC-intensive applications. For example, consider the Residu function which represents a typical filtering kernel of low bit-rate audio coding using saturating arithmetic. Implementation on XiRisc uses the PiCoGA to implement the saturating part, while the pro-

Algorithm	TI C6713				XiRisc
	Standard C (clock cycles)	C with intrinsics (clock cycles)	Optimized C (clock cycles)	Assembly-level Optimization (clock cycles)	
Residu	27910	994 (<1h.)	386 (5h.)	360 (15h.)	1914
AES	1200	-	316 (5h.)	293(20h.)	288
IDEA	360	-	373[1] (1h.)		220

(1) with an interleaving factor of 2, thus 2 blocks are elaborated concurrently

Table 7.3: XiRisc vs. TI TMS320C6713 Performance Comparison

cessor core provides the multiplier and access to the memory sub-system. This implementation requires ∼1900 clock cycles. For this kind of kernel, C6713 represents an almost application-specific architecture providing a dedicated instruction set for saturating arithmetic and 2 concurrently available multipliers. Table 7.3 reports the number of cycles required to execute Residu with respect to each improvement step. As expected, the most significant improvement is associated with introduction of the intrinsics (the dedicated instructions), which substitute both multiplication and the saturation. Nevertheless, this first step shows a result that is not so distant (only a factor 2) from the reconfigurable solution proposed by the XiRisc processor. Further optimizations have been obtained via accurate scheduling at the assembly level.

In the case of AES and IDEA, neither XiRisc nor the DSP can be classified as application-specific supports. However the multiplicative kernel of IDEA can benefit from the dual-multiplier architecture of the DSP and in the XiRisc implementation we mapped an additional multiplier on the PiCoGA.

Figure 7.4 shows a straightforward comparison of the development design spaces of XiRisc and the DSP. Performance improvements are considered for each architecture with respect to its baseline. In the case of XiRisc the baseline is the execution time of the processor core without the PiCoGA, while for the DSP it is the execution time obtained without optimizations. Speed-ups are thus normalized with respect to the basic features of a given architecture which are directly exploitable by a compiler, that is by simply writing a purely ANSI-C code. In this way we

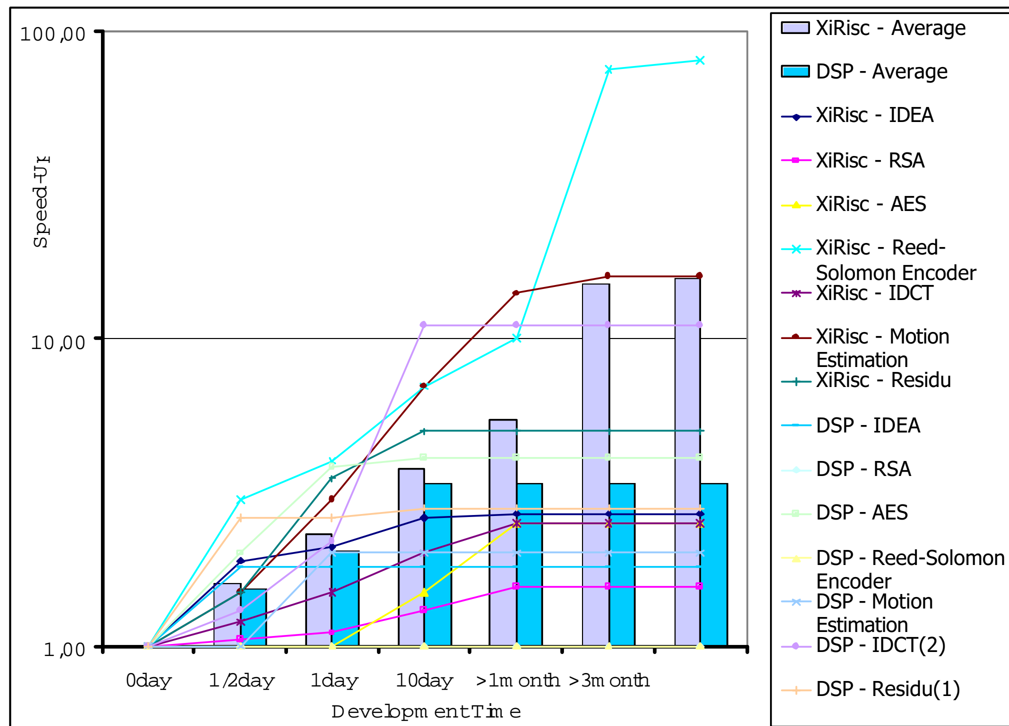


Figure 7.4: XiRisc vs DSP Development Time/Speed-Up analysis

only consider the effort required by programming the PiCoGA to accelerate a kernel regardless of the basic processor architecture where it is integrated. Only critical kernels were taken into account, because from an optimization point of view a complete application can be considered as a collection of kernels. In the comparison we considered XiRisc-efficient kernels (Reed-Solomon Encoder, Motion Estimation), DSP-efficient kernels (IDCT, Residu), and "neutral" applications (IDEA, RSA, AES). In particular, for IDCT implementation we considered implementation on a TI C5510, which uses the application-specific DCT library and hence allows one to obtain better performance, despite the additional programming complexity due to 16-bit processor architecture. We also considered implementation with intrinsics as the baseline for Residu implementation on the TI C6713, since this optimization step is performed in a very straightforward manner merely using `#define`s to rename the application-specific instructions. The average results approximatively confirm the "learning"

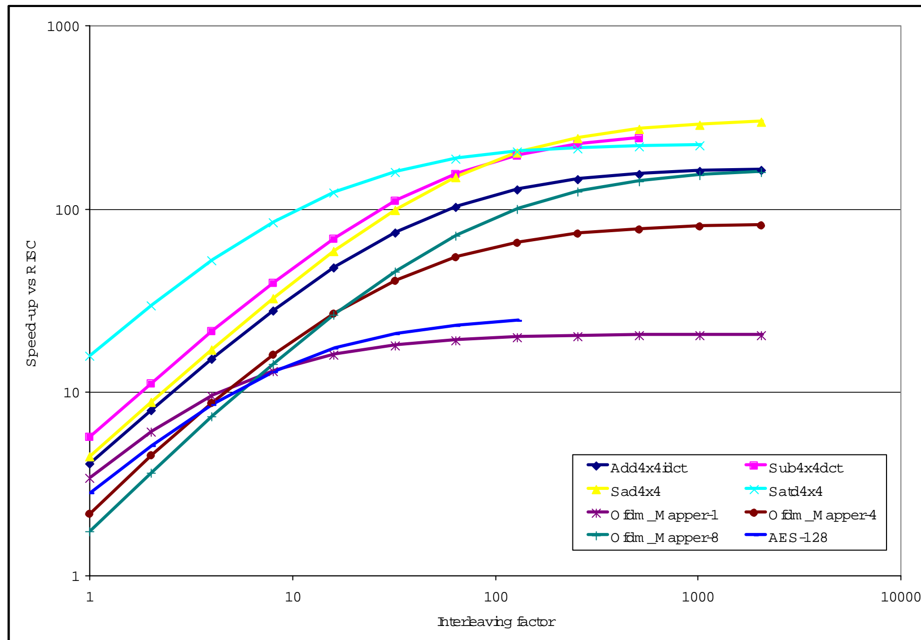


Figure 7.5: DREAM speed-up

curve of XiRisc in Figure 3.1. While the DSP curve saturates the speed-ups after a few days, XiRisc architecture allows the user to improve for a longer time, and hence achieve much higher performances.

A similar analysis can be driven in the case of the DREAM architecture. Compared to the XiRisc processor, the DREAM architecture performs instruction set extension using a co-processor model of computation. On the application side, it is possible to underline three main differences between the two processors, that could impact on both performance and development time:

- high-bandwidth direct access to the memory subsystem, by programmable address generator;
- loosely coupled register file and dedicated memory subsystem, which increases the communication overhead between processor core and reconfigurable accelerator;
- PiCoGA-III instead of PiCoGA 1.0, with roughly double computational capabilities.

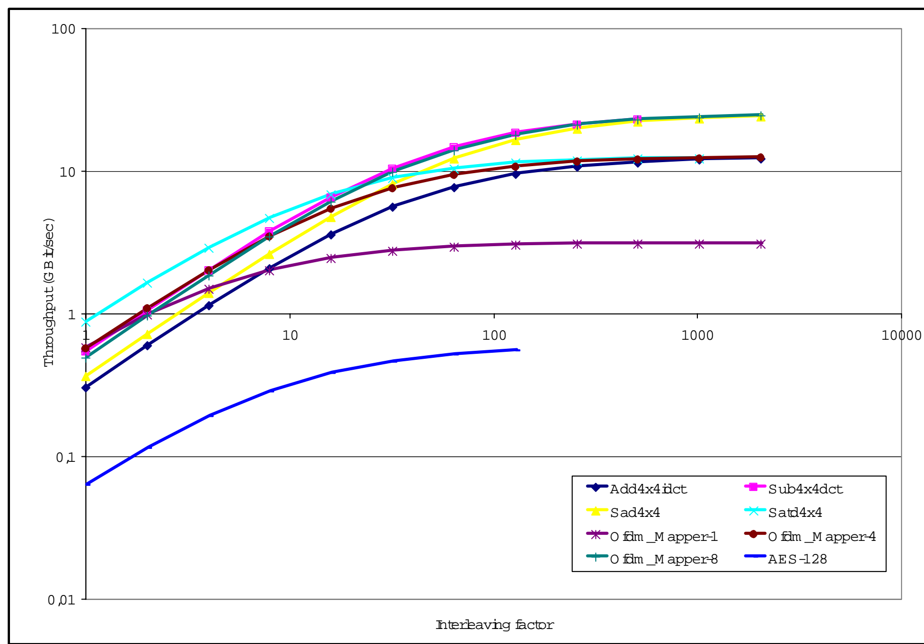


Figure 7.6: DREAM throughput

On one hand, we could apparently think that DREAM architecture is less programmable than the XiRisc processor, since the increase of communication overhead and the necessity to manually handle both the allocation of registers inside the dedicated register file and the access to the memory. But, on the other hand, these features allow DREAM to overtake most of the bottlenecks of the XiRisc processor (for example, the access to the memory), thus improving the achieved performance. It should be noticed that the memory bottleneck causes in the XiRisc processor a small utilization of the PiCoGA: 1 o 2 row are actives at times (with a peak of 12 in the motion estimation algorithm), whereas in DREAM this average value growth at 22. Performance improvements in terms of speed-up, throughput and energy efficiency for the DREAM architecture are reported respectively in Fig. 7.5, Fig. 7.6 and Fig. 7.7.

With respect to XiRisc, DREAM encourages the utilization of hardware approaches providing additional degrees of freedom, although fast software approach could benefit from the same features to overtake the increase of communication overhead. Usually, most of the time is spent

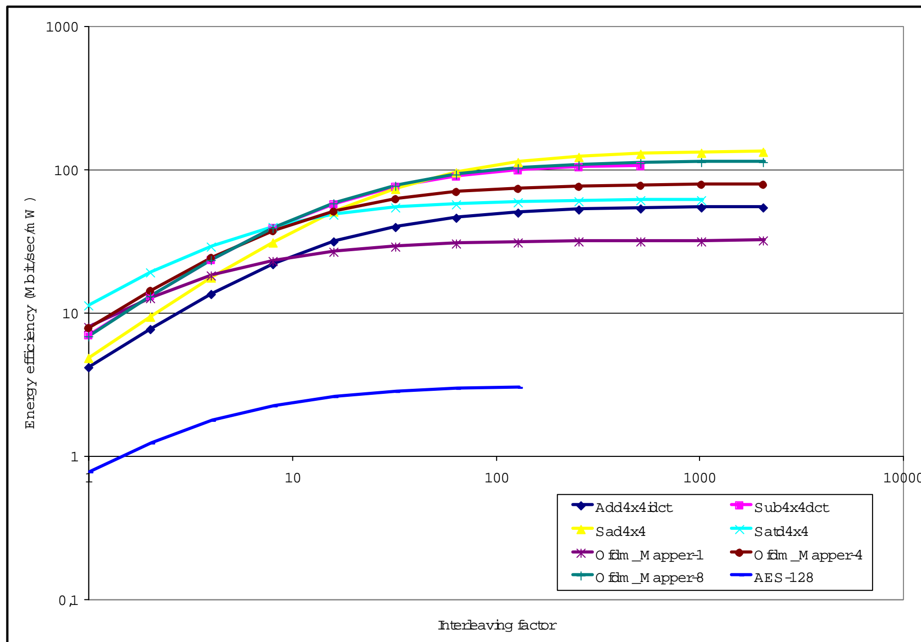


Figure 7.7: DREAM energy efficiency

on the design of the instruction set extension, deeply analyzing the algorithm and its mathematical background. In fact, following hardware approaches, DREAM achieves impressive performance in 2-4 weeks of work, whereas the XiRisc processor requires 1-2 months. Moreover, further performance improvements are easily achieved by exploiting data parallelism using address generator to perform some kind of streaming or fully pipelined computation.

Concerning the skills required to develop applications on the DREAM architecture, it is possible to note that most of the required knowledge is the same of XiRisc. In fact, the Griffy approach allows to abstract the reconfigurable device at level of software DFG, also providing the possibility to efficiently handling the pipeline structure by data dependencies. Data analysis required to exploit as much as possible the pipelining is a well known concept of DSP programming, since most of the DSPs provide scratch-pads with programmable access pattern. On this side, since it has been affirmed that embedded reconfigurable computing represents the most natural evolution of the DSP, it could said that the direct mem-

ory access provided by DREAM is not an obstacle to programmability but the solution of a bottleneck of XiRisc. For both the architectures, as well as for the XiSystem, the Griffy approach is resulted as an effective way of programming, thus proving the generality of the approach. Although XiSystem, including an additional eFPGA, is not involved in this analysis on the development time, we can reasonably expect similar results also in this case. Less speed-ups should be expected, due to communication overheads and to the utilization of a general-purpose device providing further reconfigurability at level of I/O protocols.

Chapter 8

Conclusions

During this thesis an application development environment for embedded reconfigurable processors has been developed, focusing on enabling technologies delivering reconfigurable computing to software programmers. In fact, one of the greater obstacles for the deployment of reconfigurable processors is the required co-design of both software and hardware parts. Co-design involves knowledge and skills non common in the field of embedded applications, long time dominated by solutions based on microcontrollers or DSPs augmented with application-specific hardware accelerators. Although several approaches and languages have been proposed to handle reconfigurable processors, a homogeneous and effective solution has not been found yet, mainly because of the difficulty to match performance gain with user-friendly design control. While hardware designers may obtain significant benefits from traditional HDL flows, programmers cannot optimize the software implementation without specific training.

To be appealing for the wide scenario of DSP programmers, a simplified C syntax, termed Griffy-C, has been proposed as main entry-point for the mapping on reconfigurable devices. Griffy-C provides a DFG-based abstraction of the underlying hardware, and implements the DFG in a pipelined form to increase the throughput. On the Griffy paradigm, most of the optimization steps can be driven as graph transformations (e.g. unfolding, software pipelining), and Griffy-C can be seen as the assembly-

level optimization performed by DSP developers, do not requiring any expertise on hardware design.

In the first phase of this thesis, a complete tool-chain for the XiRisc reconfigurable processor has been implemented. Griffy code is used to configure the reconfigurable functional unit of XiRisc (the PiCoGA), while the processor core is programmed by ANSI C. The implemented tool-chain provides simulation engines for both debugging and cycle-accurate performance evaluation. Graphical interfaces are provided for source-level debugging of both processor core and reconfigurable device. The user-friendly framework allowed also unexperienced users to develop their applications on the XiRisc processor obtaining valuable results after few days of work.

In a second phase, the Griffy approach has been extended to support a commercially available eFPGA, added to the XiRisc processor in the XiSystem architecture to provide pre/post processing capabilities and configurability at level of I/O protocols. A specific back-end has been implemented in order to generate VHDL from the Griffy-C. Furthermore, the Griffy approach has been applied to the DREAM adaptive DSP, including the 3rd release of PiCoGA directly connected at the memory sub-system by means of programmable address generators. In this last release, Griffy has been augmented with the capability to handle built-in functions in order to directly instance advanced operators, similarly to the case of DSP intrinsics.

A key issue of this work is a transparent instruction set extension mechanism, applied to both functional unit and co-processor models, that allows the unexperienced user full control over the embedded reconfigurable hardware, thus enabling significant performance improvements in terms of both speed and energy consumption. Applications developed, also by students, provide a quantitative assessment of this aspect, by describing precisely how much performance gain can be be obtained with a longer design cycle. In particular, two main zones can be identified depending on the programming approach and consequently the development time spent. In the first one, corresponding to pure software opti-

mization, in less than 2 days of work one can easily obtain 2-3$\times$ speed-ups simply rewriting the original code in Griffy-C, performing some memory reorganization and using standard development techniques like software pipelining and loop unrolling.

Additional development time, however, can be spent to achieve more dramatic performance improvements, up to one order of magnitude in the case of XiRisc and two orders in case of DREAM. The latter approach often requires manual optimizations, involving assembly and instruction rescheduling, deep analysis of the algorithms and their mathematical backgrounds, as well as in extreme cases hardware synthesis or hardware-aware mapping. In combination with Amdahl's law, these experimental curves, showing the different trade-offs between performance and development time, can guide application developers to identify a cost-effective mapping approaches, optimizing the overall development cost.

Appendix A

Griffy-C syntax

66. Griffy the Cooper
THE COOPER should know about tubs. / But I learned about life as well,
And you who loiter around these graves / Think you know life.
You think your eye sweeps about a wide horizon, perhaps,
In truth you are only looking around the interior of your tub.
You cannot lift yourself to its rim / And see the outer world of things,
And at the same time see yourself. / You are submerged in the tub of yourself?
Taboos and rules and appearances, / Are the staves of your tub.
Break them and dispel the witchcraft / Of thinking your tub is life!
And that you know life!

Spoon River Anthology - E. L. Masters

A.1 Overview

Griffy-C is a restricted subset of ANSI C syntax enhanced with some extensions, to handle for example variable resizing, that allow to describe software Data Flow Graph (DFG) suitable for the implementation on reconfigurable devices. Differences with other approaches reside primarily in the fact that Griffy is aimed at the extraction of a *pipelined DFG* from standard C and its mapping over a gate-array that is also pipelined by explicit stage enable signals. The fundamental feature of Griffy-based algorithm implementation is that Data Flow Control is not synthesized on the array cells but is handled separately by a hardwired control unit, thus allowing a much smaller resource utilization and easing the mapping phase. This also greatly enhances the regularity of the placing.

Griffy-C is used as a friendly format for the programming of reconfigurable devices using hand-written *behavioral descriptions* of DFGs, but can also be used as an intermediate representation (IR) automatically generated from high-level compilers. As in Fig. A.1, it is thus possible to provide different entry points for the compiling flow: high-level C descriptions, preprocessed by compiler front-end into Griffy-C, behavioral descriptions (using hand-written Griffy-C) and gate level descriptions, obtained by logical synthesis and again described at LUT level. The figure also shows the capability of programming different devices changing the back-end flow, indicating as an example the possibility to generate configurations for PiCoGA or eFPGAs (by mean of VHDL).

Restrictions essentially refer to supported operators (only operators that are significant and can benefit from hardware implementation are supported) and semantic rules introduced to simplify the mapping into the gate-array. Three basic hypothesis are assumed:

- *DFG-based description:* no control flow statements (*if*, loops or function calls) are supported, as data flow control is managed by the embedded control unit. Conditional assignments (? :) are implemented on standard multiplexers.

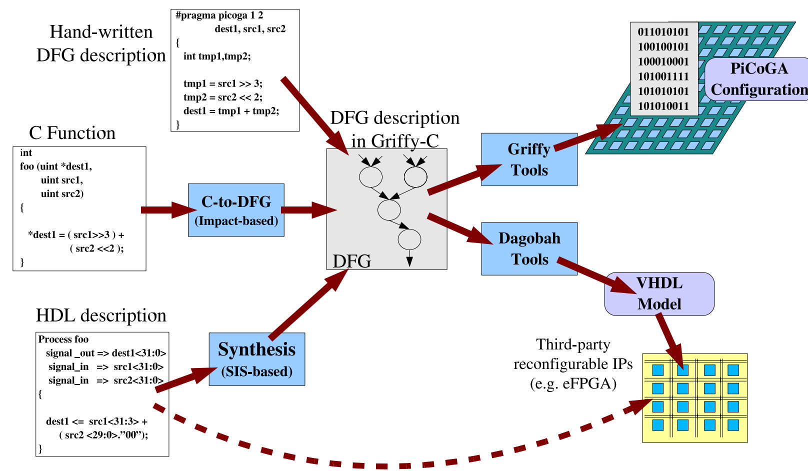


Figure A.1: Multiple entry-point Griffy flow

- *single assignment:* each variable is assigned only one time, avoiding hardware connection ambiguity.

- *manual dismantling:* only single operator expressions are allowed (similarly to intermediate representation or assembly code).

Griffy-C operators are summarized in Tab. A.1 and will be described in the following sections.

Native supported variable types are signed/unsigned int (32-bit), short int (16-bit) and char (8-bit). Width of variables can be defined at bit level using *#pragma* directives. Operators width is automatically derived from the operands size. Variables defined as *static* are used to allocate static registers inside the reconfigurable device, that is registers whose value is maintained across successive calls (i.e. to implement accumulations). All others variables are considered “local” to the operation and are not visible to successive issues.

<table>
<tr><td>Arithmetical operators</td></tr>
<tr><td>dest = src1 [+, −] src2;</td></tr>
<tr><td>Bitwise logical operators</td></tr>
<tr><td>dest = src1 [&,|,ˆ] src2; dest = ˜ src1;</td></tr>
<tr><td>Shift operators</td></tr>
<tr><td>dest = src1 [>>,<<] constant;</td></tr>
<tr><td>Comparison operators</td></tr>
<tr><td>dest = src1 [>, >=, ==, ! =, <=, <] src2;</td></tr>
<tr><td>Conditional Assignment (Multiplexer operator)</td></tr>
<tr><td>dest = src1 ? src2 : src3;</td></tr>
<tr><td>Extra-C operators</td></tr>
<tr><td>LUT operator: dest = src1 @ 0x[LUT layout];
Concatenation operator: dest = src1 # src2;</td></tr>
</table>

Table A.1: Griffy operators

Griffy-C code can be considered as a special function mapped in a reconfigurable devices, which substitutes the object code with a bit-stream. It needs a special declaration obtained with *#pragma* directive and "*picoga*" keyword. In the following, PiCoGA is considered as the target example, although the same syntax could be re-used for different platform.

```
#pragma picoga name n_outs n_ins <outs> <ins>
{
    [declaration of variables]
    [declaration of attributes]
    [PiCoGA-function body]
}
#pragma end
```

#pragma picoga syntax description:

- *name* is the name associated at the *pga-op* in the code and it can be used to call them in the ANSI C source code.
- *n_outs* is the number of outputs (no more then two) and *n_ins* is the number of inputs (no more then four).
- <*outs*> and <*ins*> are, respectively, the names of the output and input variables (separated by blanks). I/O transferring from/to the reconfigurable device is done using 32-bit *unsigned int* variables.
- *Pga-ops* declaration is closed by *#pragma* followed by "*end*" keyword.
- *Pga-ops* can be called in the ANSI C source code similarly to procedure calls as in the following example:

  ```
  name (<outs>,<ins>);
  ```

  As in ANSI C syntax, variables are separated using commas (",").
- variable declaration and function body must be inserted between curly braces ("{" and "}", as in previous example): function body cannot have more then one basic block (DFG-based description), thus no other curly braces can be used.

Variables Declaration

Variable types supported by Griffy-C syntax are the standard:

- 8-bit signed/unsigned char
- 16-bit signed/unsigned short int
- 32-bit signed/unsigned int

No array or pointer are supported. It is also defined the *static* type in order to assign a specific variable to PiCoGA registers. Similar to ANSI C, *static* variables are allocated at compiling time and exist throughout program execution (its are permanently), but their scope is the block in which they are defined (restricted visibility).

Attributes Declaration

Using *#pragma* directive with *attrib* keyword it is possible to associate some additional attributes at standard variables. For example, it is possible to define width of all variables at bit level. Syntax used to define additional attributes of the variable is the following one:

```
#pragma attrib Var1,...,VarN attribs
```

Attributes defined in Griffy-C environment are:

- *SIZE=nbits*: it sets at *nbits* the width, at bit level, of the variable; resized width must be less (or equal) then the original size: on the other hand, resizing not augments the original width of the variable.
- *SAT*: it allows to extract overflow/carryout informations from additions or subtractions that have a destination variable with SAT attribute set. Carryout and overflow are defined as additional flags of the destination variable, if and only if the operation is $+/-$. It is possible to use these flags to realize "saturation" arithmetic. Only variables that are destination of *adds* or *subs* can be used with these flags. To use these flags in *Griffy function body* is possible using special variables (implicitly defined by SAT attribute):

var_name(carryout)
var_name(overflow)

where *var_name* is the name of the variable declared SAT.

- *PIPEREG*: this flags can be used to define an explicit pipeline stage without that the variable should be seen as a *static* elements from the software environment.

Griffy-function Body

Function body description is obtained using a restricted subset of ANSI C syntax with some additional operators used to implement in the description environment grouping (or concatenation) of signals using *routing-only* resources and to implement some truth tables using the internal RLC resources (mainly, LUTs and multiplexers).

DFG-based description

Only DFG (*Data Flow Graph*) description is supported: no control statements (e.g *if* or *loops*) are defined in the Griffy syntax. The only exception is the *conditional assignment* operator that is used to implement an hardwired multiplexer. Thus, each node of a DFG can be described using a single assignment operation at Griffy-C level. When the DFG is not a DAG (*Data Acyclic Graph*) and thus it has one or more feedbacks, *static* variables must be used because, in the "software" C-compliant description of the feedback, each variable is read before written. When a feedback occurs, each *pga-ops* trigger set the value for the following *pga-ops* trigger.

Single Assignment Form

Each variable must be assigned one and only one time in the function body. This hypothesis is taken in order to avoid ambiguity and to simplify hardware translation of the DFG. In fact, under a single assignment assumption, each DFG node is unambiguously defined by the destination

variable. Dependences among nodes and instruction level parallelism is explicitly defined by the data-dependencies graph. Single assignment hypothesis is equal to assume that each variable should be used as instruction destination only one time in the DFG description. Variables can be seen as the labels of each DFG edge.

Manual Dismantling

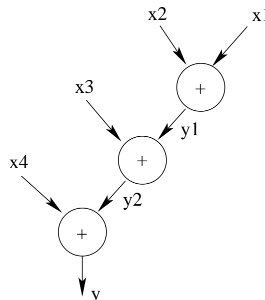
Manual Dismantling is the third assumption of Griffy-C. DFG description can be seen as an assembly language used to configure reconfigurable devices. Dismantling of complex instructions is a non-trivial task that involves instruction level parallelism (ILP) exploited in the Griffy-C description and effectively usable in the configuration flow (performed by Griffy).

For example, in order to dismantle the following complex expression:

$$y = x1 + x2 + x3 + x4;$$

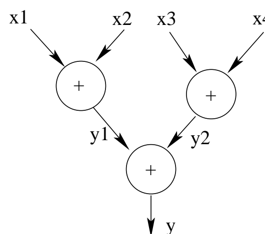
it is possible to use two main policies. The first one is a traditional *sequential* dismantling:

y1 = x1 + x2;
y2 = y1 + x3;
y = y2 + x4;



that obtains a 3 clock cycles latency and not exploit any degree of instruction level parallelism. A more efficient approach is the *balanced-tree dismantling* that exploits the most degree of instruction level parallelism. As shown in the following example, using a balanced-tree dismantling strategy, only 2 cycles of latency are required:

y1 = x1 + x2;
y2 = x3 + x4;
y = y1 + y2;



Dismantling has an important impact on both latency and issue delay of *pga-ops*.

Operator Width

Operator width is obtained involving the width of both input and output operands and type of operator. For example, adder width is set by the destination width, whereas the size of comparison is given by the greater input operand. Inputs truncation or extension are done to correctly resize the variables, padding with zeros or propagating the sign bit.

A.1.1 Standard Operators

In this section standard operators defined in Griffy-C syntax are summarized focusing on the differences (if there are) with ANSI C. Informations about the routing-only optimization are also provided. Descriptions refer to local destination variable: if destinations are *static*s or if they are attributed with *PIPEREG* flag, then routing-only optimization is not taken. Exceptions at this rule are explicitly reported in the descriptions.

A.1.2 Arithmetical Operators

Addition

Syntax:

- dest = src1 + src2;
  dest, src1, src2 are variables; width of these ones is defined by integer type or by *#pragma attrib*.
- dest = src1 + const;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted.
- dest = const + src2;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted.

- dest = const1 + const2;
  const1 and *const2* are unsigned integers; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction type is realized by constant folding and propagation: implementation do not needs RLCs.

Subtraction

Syntax:

- dest = src1 - src2;
  dest, src1, src2 are variables; width of these ones is defined by integer type or by *#pragma attrib.*

- dest = src1 - const;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted.

- dest = const - src2;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted.

- dest = const1 - const2;
  const1 and *const2* are unsigned integers; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction type is realized by constant folding and propagation: implementation do not needs RLCs.

A.1.3 Bitwise Logical Operators

Bitwise And

Syntax:

- dest = src1 & src2;
  dest, src1, src2 are variables; width of these ones is defined by integer type or by *#pragma attrib.*

- dest = src1 & const;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction masks some bit of the variable src1 and is implemented using *routing-only* resources: some bits are propagated ant the others are set to 0.

- dest = const & src2;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction masks some bit of the variable src2 and is implemented using *routing-only* resources: some bits are propagated ant the others are set to 0.

- dest = const1 & const2;
  const1 and *const2* are unsigned integers; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction type is realized by constant folding and propagation: implementation do not needs RLCs.

Bitwise Or

Syntax:

- dest = src1 | src2;
  dest, src1, src2 are variables; width of these ones is defined by integer type or by *#pragma attrib.*

- dest = src1 | const;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction is implemented using *routing-only* resources because it masks src1: some bits are propagated to destination and the others are set to 1 by constant propagation.

- dest = const | src2;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction is implemented using *routing-only* resources because it masks src2: some bits

are propagated to destination and the others are set to 1 by constant propagation.

- dest = const1 | const2;
  const1 and *const2* are unsigned integers; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction type is realized by constant folding and propagation: implementation do not needs RLCs.

Bitwise Not

Syntax:

- dest = ˜ src2;
  src2 is a variable; this instruction can be implemented manipulating destination RLCs without area occupancy.

- dest = ˜ const;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction type is realized by constant folding and propagation: implementation do not needs RLCs.

Bitwise Xor

Syntax:

- dest = src1 ˆ src2;
  dest, src1, src2 are variables; width of these ones is defined by integer type or by *#pragma attrib.*

- dest = src1 ˆ const;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction is implemented using *routing-only* resources because it masks src1: some bits are propagated to destination and the others are propagated in active low format.

- dest = const ˆ src2;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction is implemented using *routing-only* resources because it masks src2: some bits are propagated to destination and the others are propagated in active low format.

- dest = const1 ˆ const2;
  const1 and *const2* are unsigned integers; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction type is realized by constant folding and propagation: implementation do not needs RLCs.

A.1.4 Direct Assignment

Syntax:

- dest = src1;
  This instruction can be used as an explicit cast in order to resize src1 width to dest: src1 can be extended using zeros (if unsigned) or using the most significant bit (if signed) or reduced taking a slice of src1 (number of bits are defined by dest size).

- dest = const;
  This instruction is implemented using constant folding and propagation.

A.1.5 Shift Operators

Syntax:

- dest = src1 << const;
  src1 is a variable; this instruction can be implemented using routing-only resources and do not needs RLCs. Zeros are inserted as least significant bits.

- dest = const << const;
  This instruction can be implemented by constant folding and propagation.

- dest = src1 >> const;
  src1 is a variable; this instruction can be implemented using routing-only resources. When src1 is unsigned Zeros are inserted as most significant bits; when src1 is signed the most significant bit is extended.

- dest = const >> const;
  This instruction can be implemented by constant folding and propagation.

Note:
Destination variable can be used to select some bits of the source in order to realize an *unpacking*. If destination width is less then source width, then a slice of source is taken: the least significant bit is set by shift step and the most significant bit is defined by "dest_size + shift_step". For example:

```
unsigned int original_word;
unsigned char byte1, byte2, byte3, byte4;
             .......
byte1 = original_word;
byte2 = original_word >> 8;
byte3 = original_word >> 16;
byte4 = original_word >> 24;
             .......
```

can be used to unpack a word in four byte variables. This procedure must be used for unpack variables transferred from register file to PiCoGA.

A.1.6 Comparison Operators

Comparison operators must have destination variable with 1-bit *SIZE* attribute defined: Output value is 1 when comparison returns *TRUE* and 0 when comparison returns *FALSE*.

Equal

Syntax:

- dest = src1 == src2;
  dest, src1, src2 are variables; width of these ones is defined by integer type or by *#pragma attrib. SIZE* of dest must be 1.

- dest = src1 == const;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted.

- dest = const == src2;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted.

- dest = const1 == const2;
  const1 and *const2* are unsigned integers; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction type is realized by constant folding and propagation: implementation do not needs RLCs.

Not Equal

Syntax:

- dest = src1 != src2;
  dest, src1, src2 are variables; width of these ones is defined by integer type or by *#pragma attrib. SIZE* of dest must be 1.

- dest = src1 != const;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted.

- dest = const != src2;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted.

- dest = const1 != const2;
  const1 and *const2* are unsigned integers; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction type is realized by constant folding and propagation: implementation do not needs RLCs.

Less Than

Syntax:

- dest = src1 < src2;
  dest, src1, src2 are variables; width of these ones is defined by integer type or by *#pragma attrib. SIZE* of dest must be 1.

- dest = src1 < const;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. When *src1* is *signed* and *const == 0*, instruction is implemented extracting sign bit of *src1* without using RLCs.

- dest = const < src2;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted.

- dest = const1 < const2;
  const1 and *const2* are unsigned integers; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction type is realized by constant folding and propagation: implementation do not needs RLCs.

Less or Equal Than

Syntax:

- dest = src1 <= src2;
  dest, src1, src2 are variables; width of these ones is defined by integer type or by *#pragma attrib. SIZE* of dest must be 1.

- dest = src1 <= const;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted.

- dest = const <= src2;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. When *src2* is *signed* and *const == 0*, instruction is implemented extracting sign bit of *src2* and to propagate them in active low form without using dedicated RLCs.

- dest = const1 <= const2;
  const1 and *const2* are unsigned integers; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction type is realized by constant folding and propagation: implementation do not needs RLCs.

Greater Than

Syntax:

- dest = src1 > src2;
  dest, src1, src2 are variables; width of these ones is defined by integer type or by *#pragma attrib. SIZE* of dest must be 1.

- dest = src1 > const;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted.

- dest = const > src2;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. When *src2* is *signed* and *const == 0*, instruction is implemented extracting sign bit of *src2* and to propagate them without using dedicated RLCs.

- dest = const1 > const2;
  const1 and *const2* are unsigned integers; both decimal (e.g 120) and

hexadecimal (e.g. 0xfA00) formats are accepted. This instruction type is realized by constant folding and propagation: implementation do not needs RLCs.

Greater or Equal Than

Syntax:

- dest = src1 >= src2;
  dest, src1, src2 are variables; width of these ones is defined by integer type or by *#pragma attrib. SIZE* of dest must be 1.

- dest = src1 >= const;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. When *src1* is *signed* and *const == 0*, instruction is implemented extracting sign bit of *src2* and to propagate them in active low form without using dedicated RLCs.

- dest = const >= src2;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted.

- dest = const1 >= const2;
  const1 and *const2* are unsigned integers; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted. This instruction type is realized by constant folding and propagation: implementation do not needs RLCs.

A.1.7 Conditional Assignment

Conditional assignment is implemented as a multiplexing DFG node and cannot be used to explicitly perform control flow. This is the only three input edges defined in Griffy-C syntax. Variable used as conditional flag must be have *SIZE* equal to 1.

Syntax:

- dest = src1 ? src2 : src3;
  dest, src1, src2, src3 are variables; width of these ones is defined by integer type or by *#pragma attrib. SIZE* of src1 must be 1.

- dest = src1 ? src2 : const;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted.

- dest = src1 ? const : src3;
  const is an unsigned integer; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted.

- dest = src1 ? const2 : const3;
  const2 and *const2* are unsigned integers; both decimal (e.g 120) and hexadecimal (e.g. 0xfA00) formats are accepted.

- dest = const1 ? src2 : src3;
  const1 is an unsigned integer with *SIZE = 1*: implementation requires routing-only resources because conditional assignment can be resolved at compiling time; selected variable is propagated to destination without dedicated RLCs.

- dest = const1 ? const2 : src3;
  const1 is an unsigned integer with *SIZE = 1*: implementation requires routing-only resources because conditional assignment can be resolved at compiling time; src3 is propagated to destination using routing only resources if *const1* is equal to 0. Instead, if *const1* is true then *const2* is folded and propagate wherever dest is used.

- dest = const1 ? src2 : const3;
  const1 is an unsigned integer with *SIZE = 1*: implementation requires routing-only resources because conditional assignment can be resolved at compiling time; src3 is propagated to destination using routing only resources if *const1* is equal to 1. Instead, if *const1* is false then *const3* is folded and propagate wherever dest is used.

- dest = const1 ? const2 : const3;
  In this case dest is a constant calculated at compiling time: so, it is folded and propagate, similarly to other constants, wherever dest is used.

A.1.8 Advanced Operators

In this section two additional operators are shown. Its extend ANSI C syntax in order to exploit packing (concatenation) of multiple variables or to define some truth-tables implemented using RLC-only resources.

A.1.9 Concatenate operator (#)

Concatenate operator can be used in order to pack two variables into only one. If destination width is less to the sum of the widths of the sources then a packing truncated is performed.
Syntax:

- dest = src1 # src2;
  dest, src1, src2 are variables. Bits correspondence is shown in fig. A.2.

No constants can be directly used with concatenate operator. In order to concatenate a variable with a constant can be used shift and bitwise-or, or, better, using a direct assignment in order to fix the size of the constant and thus concatenate them.

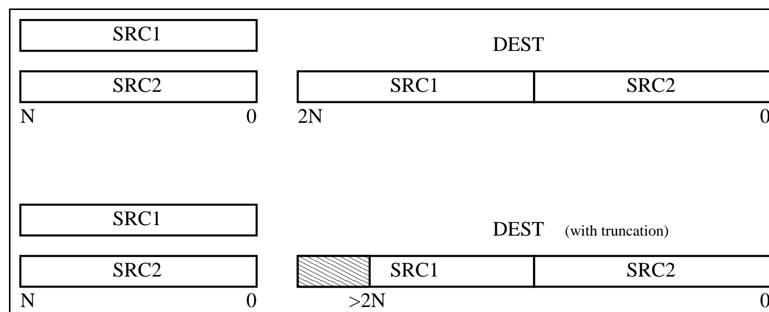


Figure A.2: Concatenate operator

A.1.10 LUT operator (@)

LUT operator (@) is defined in order to implement some typology of truth tables using resources inside single RLC. Typology of truth table is defined by destination and source width.

Syntax:

- dest = src1 @ 0x[LUT_LAYOUT];
  From destination and source is defined how to read the LUT layout (hexadecimal) string.

Typologies of supported truth tables are summarized in tab. A.2: LUTs 2x4:1 or 2x4:2 are pairs of independent LUTs, respectively 4:1 or 4:2, inside the same RLC.

SIZE attrib		LUT Typology
Src	Dest	
4	1	4:1
5	1	5:1
6	1	6:1
4	2	4:2
5	2	5:2
4	4	4:4
8	2	2 x 4:1
8	4	2 x 4:2

Table A.2: Typologies of LUTs supported

LUT Layout Rules

LUT Layout can be specified coding in an hexadecimal string the output(s) of the truth table using layout rules defined for each LUT typology. Hexadecimal string is left extended with 0 to achieve required width.

- **4:1** - **5:1** - **6:1**

SRC	DEST
0...0	♡
⋮ ⋱ ⋮	⋮
1...1	♠

dest = src @ 0x[♡...♠];

Example: and6i = src @ 0x1;

- **4:2 - 5:2**

SRC	DEST1	DEST0
0...0	♡1	♡0
⋮ ⋱ ⋮	⋮	⋮
1...1	♠1	♠0

dest = src @ 0x[♡1...♠1][♡0...♠0];
Example: and4_2 = src @ 0x00010001;
or: and4_2 = src @ 0x10001;

- **4:4**

SRC	DEST3	DEST2	DEST1	DEST0
0...0	♡3	♡2	♡1	♡0
⋮ ⋱ ⋮	⋮	⋮	⋮	⋮
1...1	♠3	♠2	♠1	♠0

dest = src @ 0x[♡3...♠3][♡2...♠2][♡1...♠1][♡0...♠0];

- **2x4:1**

SRC[7:4]	DEST1	SRC[3:0]	DEST0
0...0	♡1	*0...0*	♡0
⋮ ⋱ ⋮	⋮	⋮ ⋱ ⋮	⋮
1...1	♠1	*1...1*	♠0

dest = src @ 0x[♡1...♠1][♡0...♠0];

- **2x4:2**

SRC[7:4]	DEST3	DEST2	SRC[3:0]	DEST1	DEST0
0...0	♡3	♡2	*0...0*	♡1	♡0
⋮ ⋱ ⋮	⋮	⋮	⋮ ⋱ ⋮	⋮	⋮
1...1	♠3	♠2	*1...1*	♠1	♠0

dest = src @ 0x[♡3...♠3][♡2...♠2][♡1...♠1][♡0...♠0];

A.1.11 Built-in function as hard-macros

Reconfigurable devices featuring basic logic block with advanced operators can benefit of the direct instance of these functionalities. The case is analogue to the built-in function provided in DSP processor in order to explicitly instance some assembly instruction, like a sum of absolute differences or a saturating sum. The syntax adopted is the same of standard function calls, although the number of inputs depends on the specific functionality.
Syntax:

- dest = my_hard_macro (src1, ..., srcN);

Example: PiCoGA-III specific built-in functions

PiCoGA-III features a hybrid reconfigurable logic cells allowing straightforward implementations of non-standard operations by mean of dedicated logic. As an example, the simple adder is implemented using the ALU block, but more complex operations can be provided coupling ALU features with the surrounding control logic. To give an idea of that capabilities, the following items shows a basic set of built-in functions already implemented on the PiCoGA-III specific Griffy flow.

GFMult : implements the multiplication on the Galois Field GF(2^4) with the irreducible polynomial $x^4 + x + 1$;

$$dest = GFMult\ (src1, src2);$$

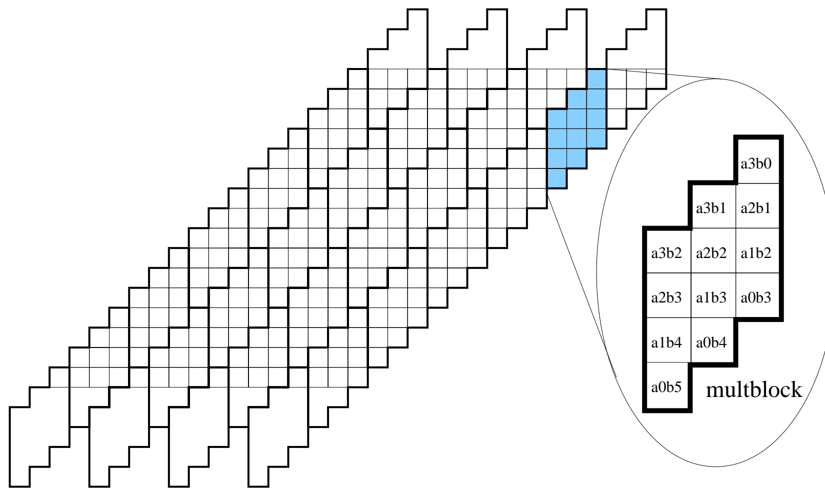


Figure A.3: Multiplier chunk

multblock : implements a multiplier chunk as shown in Fig. A.3

$$dest = multblock\ (src1, src2);$$

CondSum : implements a conditional sum. Depending on a condition flag, it performs a sum or a subtraction between two operands

$$dest = CondSum\ (cond, src1, src2);$$
$$\rightarrow \quad dest = cond?src1 - src2\ :\ src1 + src2;$$

Accumulator : implements a routing-efficient accumulator in which the feedback path is internal to the reconfigurable logic cell

$$dest = Accumulator(src);$$
$$\rightarrow \quad dest = dest + src;$$

CondAccumulator : implements a routing efficient conditional accumulator which accumulates or reset to a constant depending on the condition

$$dest = CondAccumulator(cond, const, src);$$
$$\rightarrow \quad dest = src + (cond?const : dest);$$

Xor10bit : implements a single-cell xor among 10 bits

SuperMux : implements a 4-input 1-output multiplexer, based on 2-bitwise chunk implemented on a single cell.

Bibliography

[1] J. Rabaey *Reconfigurable Computing: The solution to Low Power Programmable DSP*, Proceedings of the 1997 IEEE International Conference on Acoustics, Speech and Signal Processing, April 1997.

[2] W.H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V.K. Prasanna, H.A.E. Spaanenburg *Seeking solutions in configurable computing*, IEEE Computer, Dec 1997.

[3] J. Goodacre, A.N. Sloss *Parallelism and the ARM Instruction Set Architecture*, IEEE Computer, May 2005.

[4] S. Leibson, J. Kim *Configurable Processors: a new era in chip design*, IEEE Computer, May 2005.

[5] V.Kathail, S.Aditya, R.Schreiber, B.R.Rau, D.C.Cronquist, M.Sivaraman *PICO: Automatically Designing Custom Computers*, IEEE Computer, Feb 2002.

[6] N.T. Clark, H. Zhong, S.A. Mahlke *Automated Custom Instruction Generation for Domain-Specific Processor Acceleration*, IEEE Transactions on Computers, Vol. 54, No. 10, October 2005.

[7] A. DeHon *The density advantage of configurable computing*, IEEE Computer, April 2000.

[8] K. Bondalapati, V.K. Prasanna *Reconfigurable Computing Systems*, Proceedings of the IEEE, Vol. 90, No. 7, April 2002.

[9] L. B. Baumstark, L. M. Wills *Retargeting Sequential Image-Processing Programs for Data Parallel Execution*, IEEE Transactions on Software Engineering, vol. 31, no. 2, February 2005.

[10] F. Barat, R. Lauwereins, G. Deconinck *Reconfigurable instruction set processors from a hardware/software perspective*, IEEE Transactions on Software Engineering, Volume 28, Issue 9, Sept. 2002.

[11] A. DeHon, J. Wawrzynek *Reconfigurable Computing: What, Why and Implications for Design Automation*, Proceeding on DAC, 1999.

[12] S. Vassiliadis, S. Wong, S. Gaydadjiev, K. Bertels, G. Kuzmanov, E.M. Panainte *The MOLEN Polymorphic Processor*, IEEE Transactions on Computers, Nov. 2004.

[13] B. Mei, S. Vernalde, D. Verkest, H. De Man, R. Lauwereins *DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architecture*, Int l Conference on Field Programmable Technology, Dec. 2002.

[14] Trimaran consortium. The Trimaran Compiler Infrastructure, *http://www.trimaran.org*.

[15] C. Mucci, F. Campi, A. Deledda, A. Fazzi, M. Ferri, M. Bocchi A cycle-accurate ISS for a dynamically reconfigurable processor architecture, IEEE Reconfigurable Architecture Workshop (RAW), Apr. 2005

[16] P.M. Athanas, H.F. Silverman *Processor Reconfiguration Through Instruction-Set Metamorphosis*, IEEE Computer, March 1993.

[17] R. Razdan; M.D. Smith *A High-Performance Microarchitecture with Hardware-Programmable Functional Units*, Proceedings of IEEE MICRO, Nov. 1994.

[18] R.D. Wittig; P. Chow *OneChip: An FPGA Processor With Reconfigurable Logic*, IEEE Symposium on FPGA for Custom Computing Machine, 1996.

[19] T.J. Callahan, J.R. Hauser, J. Wawrzynek *The Garp architecture and C compiler*, IEEE Computer, April 2000.

[20] J-Y. Mignolet, V. Nollet, P. Coene, D.Verkest, S. Vernalde, R. Lauwereins *Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip*, Proceedings of the DATE 2003.

[21] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Water, W.W. Hwu *IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors*, Proceedings of the 18th Annual Int'l Symposium on Computer Architecture, Toronto, Canada, May 28, 1991, pp. 266-275

[22] SUIF Compiler System [online] *http://suif.standford.edu*

[23] J.M. Arnold *S5: the architecture and development flow of a software configurable processor*, Proceedings on IEEE International Conference on Field-Programmable Technology, 11-14 Dec. 2005, pp. 121-128

[24] Sato, T.; Watanabe, H.; Shiba, K.; *Implementation of dynamically reconfigurable processor DAPDNA-2*, IEEE VLSI-TSA International Symposium VLSI Design, Automation and Test, 27-29 April 2005, pp. 323-324

[25] R. Baines and D. Pulley *A Total Cost Approach to Evaluating Different Reconfigurable Architectures for Baseband Processing in Wireless Receivers*, IEEE Communication Magazine, Jan. 2003.

[26] Elixent *http://www.elixent.com*

[27] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, B. Hutchings *A reconfigurable arithmetic array for multimedia applications* Proceedings of the ACM/SIGDA International Symposium on Field programmable gate arrays (FPGA), 1999.

[28] Hennessy, Patterson *Computing architecture: a quantitative approach*, Morgan Kaufmann

[29] A. DeHon, J. Adams, M. DeLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, M. Wrighton *Design Patterns for Reconfigurable Computing*, IEEE Symposium on FCCM, April 2004.

[30] R. Hartenstein *A Decade of Reconfigurable Computing: a Visionary Retrospective*, DATE 2001.

[31] F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic Publishers, 1997.

[32] L.M. Reynari, F. Cucinotta, A. Serra, L. Lavagno *A hardware/software co-design flow and IP library based of Simulink*^{TM}, Proceedings on DAC, Jun. 2001.

[33] A. Baghdadi, N.-E. Zergainoh, W. O. Cesario, A.A. Jerraya, *Combining a Performance Estimation Methodology with a Hardware/Software Codesign Flow Supporting Multiprocessor Systems*, IEEE Transactions on Software Engineering, vol. 28, no. 9, September 2002.

[34] *http://www.systemc.org/*

[35] A. Gerstlauer, R. Dmer, P.Junyu, D.D. Gajski, *System Design: A Practical Guide with SpecC*, Kluwer Academic Publishers, 2001.

[36] Sullivan, C.; Wilson, A.; Chappell, S.; *Using C based logic synthesis to bridge the productivity gap*, Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC), 27-30 Jan. 2004, pp. 349-354

[37] J. Frigo, M. Gokhale, D. Lavenier *Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective*, Proceeding on FPGA 2001.

[38] S. Gupta, N. Dutt, R. Gupta, A. Nicolau *SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations*, International Conference on VLSI Design, January 2003.

[39] G. De Micheli *Hardware synthesis from C/C++ models*, Proceedings of Design, Automation, and Test in Europe (DATE), Munich, Germany, March 1999.

[40] W.A. Najjar, W. Bohm, B.A. Draper, J. Hammes, R. Rinker, J.R. Beveridge, M. Chawathe, C. Ross *High-Level Language Abstraction for Reconfigurable Computing*, IEEE Computer, August 2003.

[41] S. Edwards *The Challenges of Hardware Synthesis from C-like Languages*, Proceedings on IWLS 2004.

[42] K. Keutzer, S. Malik, A.R. Newton, J.M. Rabaey, A. Sangiovanni-Vincentelli *System-Level Design: Orthogonalization of Concerns and Platform-Based Design*, IEEE Transactions on CAD, Dec. 2000

[43] L. Semeria, K. Sato, G. De Micheli *Synthesis of hardware models in C with pointers and complex data structures*, IEEE Transactions on VLSI Systems, Dec. 2001

[44] R. Camposano, W. Rosenstiel *Synthesizing Circuits From Behavioral Descriptions*, IEEE Transactions on Computer-Aided Design, February 1989.

[45] R. Camposano *From Behavior to Structure: High-Level Synthesis*, IEEE Design & Test of Computers, October 1990.

[46] S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. Reed Taylor *PipeRench: A Reconfigurable Architecture and Compiler*, IEEE Computer, April 2000.

[47] M. Budiu, S.C. Goldstein *Fast Compilation for Pipelined Reconfigurable Fabrics*, FPGA 1999.

[48] D.C. Cronquist, P.Franklin, S.G. Berg, C. Ebeling *Specifying and Compiling Applications for RaPiD*, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (FCCM), April 1998.

[49] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe and Anant Agarwal *The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs*, IEEE Micro, Mar/Apr 2002.

[50] M.B. Gokhale, J.M. Stone *NAPA C: Compiling for a Hybrid RISC/FPGA Architecture*, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (FCCM), April 1998.

[51] S. Talla *Adaptative Explicit Parallel Instruction Computing*, PhD Thesis, Department of Computer Science, New York University, May 2001.

[52] V.S. Gheorghita, W.-F. Wong, T. Mitra, S. Talla *A Co-simulation Study of Adaptative EPIC Computing*, Proceedings on IEEE Field Programmable Technologies (FPT), 2002.

[53] M. Vorbach, J. Becker, *Reconfigurable processor architectures for mobile phones* Proceedings on the Int'l Parallel and Distributed Processing Symposium, 22-26 April 2003.

[54] Florian Stock, Andreas Koch *Architecture Exploration and tools for pipelined coarse grained reconfigurable arrays*, Proceedings on the IEEE Int l Conference on Field Programmable Logic and Application (FPL), Aug. 2006.

[55] H. Singh, M.-H. Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, E.M. Chaves Filho *MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications*, IEEE Transactions on Computers, May 2000.

[56] C. Rowen, S. Leibson *Engineering the Complex SOC: Fast, Flexible Design with Configurable Processors*, Prentice-Hall, 2004.

[57] C. Lee, M. Potkonjak, W.H. Mangione-Smith *MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems*,

30th Annual International Symposium on Microarchitecture (Micro '97), December 1997.

[58] R. Lysecky, F. Vahid *A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning*, Proceedings on the Design Automation and Test in Europe Conference (DATE), February 2004.

[59] G. Snider *Performance-Constrained Pipelining of Software Loops onto Reconfigurable Hardware*, Proceeding on FPGA 2002.

[60] M. Weinhardt and W. Luk *Pipeline Vectorization*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Feb. 2001, pp. 234-248.

[61] V. Allan, R. Jones, R. Lee, S. Allan *Software Pipelining*, ACM Computing Surveys, Vol. 27, No. 3 September 1995.

[62] B. R. Rau. *Iterative Modulo Scheduling*. Technical Report HPL-94-115, Hewlett Packard Company, November 1995.

[63] A.H. Veen *Dataflow Machine Architecture*, ACM Computing Surveys, Vol. 18, No. 4, December 1986.

[64] G. Gao, Y. Wong, Q. Ning *A Timed Petri-Net Model for Fine-Grain Loop Scheduling*, Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, June 1991.

[65] M.Mukund *Petri Nets and Step Transition Systems*. International Journal of Foundations of Computer Science 3, 443-478, World Scientific, 1992.

[66] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, R. Guerrieri *A VLIW processor with reconfigurable instruction set for embedded applications*, IEEE Journal of Solid-State Circuit, Nov. 2003.

[67] A. Lodi, A. Cappelli, M. Bocchi, C. Mucci, M. Innocenti, C. De Bartolomeis, L. Ciccarelli, R. Giansante, A. Deledda, F. Campi, M. Toma

and R. Guerrieri, *XiSystem: a XiRisc-based SoC with a Reconfigurable IO module*, IEEE Journal of Solid-State Circuit (JSSC), Jan. 2006

[68] M. Bocchi, C. De Bartolomeis, C. Mucci, F. Campi, A. Lodi, M. Toma, R. Canegallo, R. Guerrieri *A XiRisc-based SoC for Embedded DSP Applications*, IEEE Custom Integrated Circuits Conferences (CICC'04), Oct. 2004

[69] A. Lodi, M. Toma, F. Campi *A Pipelined Configurable Gate Array for Embedded Processors*, Proceeding on FPGA 2003.

[70] A. Cappelli, A. Lodi, C. Mucci, M. Toma, F. Campi *A Dataflow Control Unit for C-to-Configurable Pipelines Compilation Flow*, IEEE Symposium on FCCM, Apr. 2004.

[71] C. Mucci, C. Chiesa, A. Lodi, M. Toma, F. Campi *A C-based Algorithm Development Flow for a Reconfigurable Processor Architecture*, IEEE International Symposium on System on Chip, November 2003.

[72] C. Mucci, F. Campi, A. Deledda, A. Fazzi, M. Ferri, M. Bocchi *A cycle-accurate ISS for a dynamically reconfigurable processor architecture*, IEEE Reconfigurable Architecture Workshop (RAW), Apr. 2005.

[73] A. La Rosa, L. Lavagno, C. Passerone, *Software Development for High-Performance, Reconfigurable, Embedded Multimedia Systems*, IEEE Design and Test of Computers, vol. 22, no. 1, pp. 28-38, January/February, 2005.

[74] C. Mucci, M. Bocchi, P. Gagliardi, L. Ciccarelli, A. Lodi, M. Toma, F. Campi *A Case-Study on Multimedia Applications for the XiRisc Reconfigurable Processor*, Proceedings on IEEE Int'l Symposium on Circuits and Systems (ISCAS), May 2006.

[75] F. Campi, A. Deledda, M. Pizzotti, L. Ciccarelli, C. Mucci, A. Lodi, A. Vitkovski, L. Vanzolini, P. Rolandi *A dynamically adaptive DSP for heterogeneous reconfigurable platforms*, Proceedings on IEEE/ACM DATE 2007.

[76] E. Sentovich et al. *SIS: A System for Sequential Circuit Synthesis*, UCB/ERL M92/41, May 1992.

[77] D. Xu, X. He, Y.Deng, *Compositional Schedulability Analysis of Real-Time Systems Using Time Petri Nets*, IEEE Transactions on Software Engineering, vol. 28, no. 10, October 2002.

[78] A. Koch *Module Compaction in FPGA-based Regular Datapaths*, Proceeding on DAC 1996.

[79] T.J. Callahan , P. Chong, A. DeHon, and J. Wawrzynek *Fast Module Mapping and Placement for Datapaths in FPGAs*, Proceeding on FPGA 1998.

[80] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, J. Stockwood *Hardware-Software Co-Design of Embedded Reconfigurable Architectures*, Proceedings on DAC, 2000.

[81] V. Betz, J. Rose, A. Marquardt *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, 1999.

[82] *COCOMO 2.0 Model Definition manual*, ver 1.2, 1997.

[83] Capers Jones, Chairman, Software Productivity Research, Inc. *Programming Languages Table*, Release 8.2, March 1996. *http://www.theadvisors.com/langcomparison.htm*

[84] J. A. Debardelaben, V. K. Madisetti, A. J. Gadient *Incorporating Cost Modeling in Embedded-System Design*, IEEE Design & Test of Computer, Vol. 14, Issue 3, pag. 24-35, July-Sept. 1997

[85] D. Ragan, P. Sandborn, P. Stoaks *A Detailed Cost Model for Concurrent Use With Hardware/Software Co-Design*, Proceedings on the Design Automation Conference (DAC), June 10-14, 2002, New Orleans, LA.

[86] W. Fornaciari, F. Salice, U. Bondi, E. Magini, *Development Cost and Size Estimation Starting from High-Level Specifications*, Proceedings on the International Symposium on Hardware/Software Codesign (CODES), April 25-27, 2001, Copenhagen (Denmark).

[87] V. K. Madisetti, J. A. Debardelaben *A RASSP Approach to HW/SW Codesign*, RASSP Digest, Vol. 2 4th Qtr. 1995.

[88] A. La Rosa, L. Lavagno, M. Lazarescu, C. Passerone, *An optimizing C front-end for hardware synthesis*, Proceedings on the workshop on Wireless Reconfigurable Terminals and Platforms (WiRTeP), April, 10-12, 2006, Rome (Italy).

[89] D. Burger, T. Austin *The SimpleScalar Tool Set, Version 2.0*, www.simplescalar.com

[90] S. Pees, A. Hoffmann, V. Zivojnovic, H. Meyr *LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures*, Proceedings on DAC, Jun. 1999.

[91] J. Cardillo, P. Chow *The Effect of Reconfigurable Units in Superscalar Processors*, Proceedings on FPGA, February 2001.

[92] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, E.M. Panainte *The MOLEN Polymorphic Processor*, IEEE Transactions on Computers, November 2004

[93] Suddep Pasricha *Transaction level modeling of Soc with SystemC 2.0*, Synopsys User Group Conference (SNUG), 2002.

[94] M. Caldari, M. Conti, M. Coppola, S. Curaba, L. Pieralisi, C. Turchetti *Transaction-Level Model for AMBA Bus Architecture Using SystemC 2.0*, Proceedings of the Design,Automation and Test in Europe Conference and Exhibition (DATE 2003)

[95] L. Di Stefano, S. Mattoccia, F. Tombari *Speeding-up NCC-based template matching using parallel multimedia instructions*, Proceedings on the 7th Int'l Workshop on Computer Architecture for Machine Perception, 4-6 July 2005.

[96] Texas Instruments Incorporated. TMS320C6713, TMS320C6713B Floating-point Digital Signal Processors. [Online]. Available: *http://focus.ti.com/lit/ds/symlink/tms320c6713.pdf*

[97] GNU Compiler Collection (GCC) [online available] *http://gcc.gnu.org*

[98] C. Brunelli, F. Garzia, F. Campi, C. Mucci, J. Nurmi *A FPGA Implementation of an Open-Source Floating-Point Computation System*, IEEE Int l Symposium of System-on- Chip, Tampere (Finland), Nov. 2005.

[99] T. Sikora, *MPEG Digital Video-Coding Standards*, IEEE Signal Processing Magazine, September 1997.

[100] MPEG Software Simulation Group *http://www.mpeg.org*

[101] ISO/IEC 13818 Draft International Standard: Generic Coding of Moving Pictures and Associated Audio, Part-2: video.

[102] A. Dasu and S. Panchanathan *A survey of media processing approaches*, IEEE Transactions on Circuits and Systems for Video Technology, August 2002.

[103] F. Campi et al. *A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications*, ISSCC 2003.

[104] P.L. Tai, S.Y. Huang, C.T. Liu and J.S. Wang *Computation-Aware Scheme for Software-Based Block Motion Estimation*, IEEE Transactions on Circuits and Systems for Video Technology, September 2003

[105] C. De Vleeschouwer, T. Nilsson, K. Denolf, J. Bormans *Algorithmic and Architectural Co-Design of a Motion-Estimation Engine for Low-Power Video Devices*, IEEE Transactions on Circuits and Systems for Video Technology, December 2002

[106] NIST *Specification for the ADVANCED ENCRYPTION STANDARD (AES)*, FIPS PUBS 197, November 26, 2001.

[107] J. Daemen and V. Rijmen *AES Proposal: Rijndael*, NIST AES Proposal, www.esat.kuleuven.ac.be/˜rijmen/rijndael/.

[108] B. Schneier, *Applied Cryptography*, 2nd ed. John Wiley and Sons. New York, NY, 1996.

[109] S. Ravi, A. Raghunathan, N. Potlapally, M. Sankaradass *System Design Methodologies for a Wireless Security Processing Platform*, Proceedings on the DAC 2002.

[110] S. Tillich et al. *An Instruction Set Extension for Fast and Memory-Efficient AES Implementation*, Communications and Multimedia Security, Springer Verlag, 2005.

[111] M2000 Inc. *www.m2000.fr*

[112] T. Wollinger, M. Wang, J. Guajardo, C. Paar *How well Are High-End DSPs Suited for the AES Algorithms? (AES Algorithms on the TMS320C6x DSP)* Presentation at the NIST AES-3 Conference, 2000. *http://csrc.nist.gov/CryptoToolkit/aes/round2/conf3/presentations/wollinger.pdf*

[113] J. Zambreno et al. *Exploring Area/Delay Tradeoffs in an AES FPGA Implementation*, FPL 2004.

[114] R. Chaves et al. *Reconfigurable Memory Based AES Co-Processor*, Proceedings of the RAW, April 2006

[115] HELION. *High Performance AES (Rijndael) cores for Xilinx FPGA*, http://www.heliontech.com

[116] A. Wiesmaier, *The State of the Art in Algorithmic Encryption (2006)*, citeseer.ist.psu.edu/wiesmaier06state.html

[117] A. Hodjat and I. Verbauwhede *A 21.54 Gbit/s fully pipelined AES processor on FPGA*, FCCM 2004.

[118] J. Lu, J. Lockwood *IPSec Implementation on Xilinx Virtex-II Pro FPGA and Its Application*, RAW 2005

[119] P.R. Schaumont, H. Kuo, I. Verbauwhede *Unlocking the Design Secrets of a 2.29 Gb/s Rijndael Processor*, DAC 2002.

[120] R.R. Taylor, S.C. Goldstein *A High-Performance Flexible Architecture for Cryptography*, CHES 1999.

[121] C. Paar *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*, Ph.D. Thesis, 1994.

[122] V. Rijmen *Efficient Implementation of the Rijndael S-box*,

[123] G. Bertoni et al. *Efficient Software Implementation of AES on 32-Bit Platforms*, CHES 2002.

[124] T. Wiegand, G.J. Sullivan, G. Bjntegaard, A. Luthra *Overview of the H.264/AVC video coding standard*, IEEE Transaction on Circuits and Systems for Video Technology, July 2003.

[125] H. S. Malvar, A. Hallapuro, M. Karczewicz, L. Kerofsky *Low-Complexity Transform and Quantization in H.264/AVC*, IEEE Transaction on Circuits and Systems for Video Technology, July 2003.

[126] Yu-Wen Huang, Bing-Yu Hsieh, Tung-Chien Chen, Liang-Gee Chen *Analysis, fast algorithm, and VLSI architecture design for H.264/AVC intra frame coder* IEEE Transactions on Circuits and Systems for Video Technology, March 2005.